Ambiente Unix

Maurizio Matteuzzi
Paolo Pellizzardi

Ambiente Unix

Maurizio Matteuzzi
Paolo Pellizzardi

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

SOMMARIO

PREMESSA

Il sistema operativo UNIX rappresenta sotto molti aspetti un oggetto singolare nella storia dell'informatica: pur con una struttura che per certi versi potremmo definire relativamente semplice, esso porge molte funzionalitá paragonabili a quelle dei piú sofisticati sistemi operativi da mainframe. Per un altro verso la duttilitá e la modularitá dei comandi, e la stessa strategia generale di affrontare situazioni complesse attraverso insiemi di programmi piú semplici possibile, modulari e concatenabili, anziché con specifici e complicati programmi ad hoc, fanno di UNIX uno strumento che richiede un training relativamente breve per cominciare ad usarlo, quanto meno nelle applicazioni piú elementari. Queste caratteristiche riflettono le due componenti essenziali della vicenda UNIX: l'ambiente dello sviluppo del software, le cui finalitá UNIX rispecchia marcatamente, e quello dell'attivitá universitaria, sia didattica che di ricerca. Infatti é ben noto che UNIX é il sistema operativo in assoluto piú popolare in ambiente universitario. In linea generale, prescindendo dal punto di vista strettamente tecnico, si puó dire che il successo di UNIX é dovuto in buona parte al giusto dosaggio di queste due componenti. Ci é sembrato dunque opportuno, per realizzare un libro su UNIX, cercare di riprodurre le dosi della ricetta fortunata, miscelando la competenza accademica con la ricerca attiva nel mondo dell'industria.

Il testo si avvale dell'esperienza di laboratorio della West 80 di Bologna, societá che da piú di tre anni sviluppa applicazioni di automazione d'ufficio in ambiente UNIX.

Gli Autori ringraziano amici e colleghi della West 80, e in particolare Stefano Fabi, che ha curato la tavola riepilogativa dei comandi di uso piú frequente posta in appendice al volume, e Valerio Carioli, che ha messo a disposizione la sua preziosa competenza sugli aspetti piú sofisticati del sistema UNIX.

CAPITOLO I

UNO SGUARDO COMPLESSIVO

1.1 STORIA DI UNIX

L'idea di UNIX nasce verso la fine degli anni '60 nell'ambiente dei laboratori di ricerca della **Bell** , del gruppo AT&T , quelli stessi ben noti nel mondo della tecnica per gli straordinari risultati ottenuti nel campo dei semiconduttori (a tutt'oggi detentori del brevetto del **laser** e del **transistor**), ad opera di **Ken Thompson.** Questi si ispira ad alcuni concetti del **Multics** , sistema operativo **time-sharing** progettato al MIT per grossi sistemi, con la collaborazione dei Bell Laboratories. L'idea iniziale é quella di costruire un sistema operativo **single user** (da cui il nome) particolarmente adatto e flessibile per lo sviluppo del software. Nasce cosí nel 1969 la prima versione di UNIX, scritta in assembler, che gira sui minicomputer PDP7 e PDP9 della Digital. Data la bontá del primo prodotto, l'esperienza fu potenziata, e nacque ben presto una versione **multi user** sui mini della linea PDP11 sempre della Digital. Questo passo segna una svolta decisiva nel destino di UNIX. Infatti la nuova versione multi user non é piú scritta in assembler come la precedente, ma per la gran parte, in linguaggio **C.** Il **C** é a sua volta un prodotto dei Bell Laboratories, e nasce in quegli stessi anni ad opera di D.M.Ritchie come sviluppo del "B", altra creatura di Thompson, linguaggio orientato allo sviluppo del software di base. Questo fatto rappresenta per UNIX un evento determinante. Viene concepita infatti l'idea di riscrivere il **compilatore** del linguaggio **C** in **C** stesso. Ció segna un primo passo nell'ordine di idee di rendere UNIX un prodotto trasportabile, in larga parte indipendente dalla particolare macchina di cui é ospite: il compilatore di **C** infatti é in grado di rigenerare l'UNIX per ogni nuovo elaboratore, una volta adattate le routines semantiche al relativo codice macchina, il che limita fortemente la parte da riscrivere.

Il fatto che la stesura di UNIX sia dovuta principalmente a due sole persone, Thompson e Ritchie, garantisce al sistema una particolare compattezza, e una forte coerenza interna rispetto alle idee ispiratrici fondamentali; un po' come, per citare un altro esempio famoso, il fatto che il PASCAL sia stato concepito ed elaborato da una sola persona, N. Wirth, gli fornisce un grado di compattezza che altri linguaggi non hanno.
La grande trasportabilitá di UNIX e la circostanza che la Bell non era presente direttamente nel settore dell'informatica, e non aveva quindi interesse all'esclusivitá del prodotto, hanno gettato le basi per l'enorme sviluppo successivo. In primo luogo, esso ha avuto uno strepitoso successo nel campo universitario. E questa é la seconda svolta fondamentale della sua storia. Infatti, seguendo la politica di renderlo disponibile quasi gratuitamente al mondo universitario, la Bell ha ottenuto di coinvolgere nello sviluppo di UNIX un altro ambiente di ricerca, della massima potenzialitá. Ne ha ottenuto in breve un progressivo miglioramento del prodotto, e, piú ancora, un grande arricchimento di software strettamente correlato a UNIX, e la diffusione del prodotto presso vaste schiere di giovani laureati.

Se il primo "mondo" aggredito da UNIX é stato quello delle universitá, il secondo é stato quello dei produttori di elaboratori della Silicon Valley. E' ben noto che, in seguito allo produzione e alla larga e rapida diffusione dei microprocessori, é venuto via via crescendo il numero delle aziende di piccole e medie dimensioni produttrici di microelaboratori; questo fenomeno si é manifestato in modo particolarmente appariscente in certe aree geografiche, come la California settentrionale a sud di S. Francisco. Quando, dalla fase iniziale dei sistemi monoutente, prevalentemente con sistema operativo CP/M , il mercato ha cominciato a spostarsi verso i sistemi multiutente, si é creato in modo impellente il problema di un sistema operativo multi user; la scelta di UNIX é apparsa a molti quasi obbligata, data la relativamente facile trasportabilitá, e il conseguente abbattimento dei tempi necessari a licenziare in via definitiva un sistema con un software di base sufficientemente affidabile e con buone performance.

1.2 LA DISTRIBUZIONE LE LICENZE

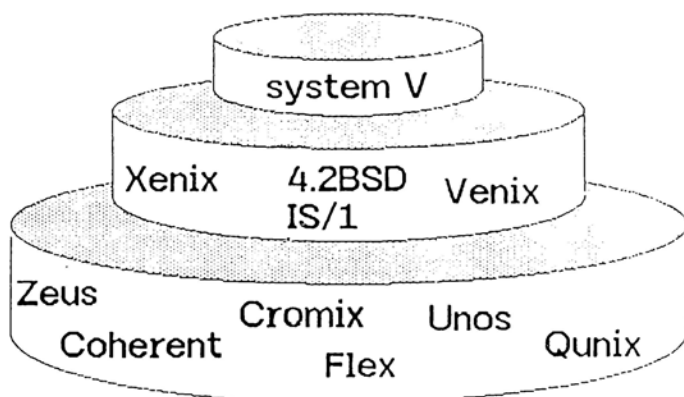
In seguito alle successive implementazioni, cui si é fatto cenno, si é quindi arrivati al settimo livello di UNIX. La Bell non lo distribuiva direttamente ma attraverso la **Western Electric** , consorella del gruppo **AT&T.** La politica di vendita delle licenze adottata dalla Bell é stata fortemente diversificata. La scelta fondamentale della Bell fu di praticare prezzi fortemente diversi a seconda che si trattasse di clienti industriali o No Profit Organization (tipicamente Università).

Fino a poco tempo fa la Bell vendeva il prodotto senza assumersi alcuna responsabilità di manutenzione o assistenza. Non essendo presente con propri elaboratori sul mercato, era poco affetta da tutta la problematica tipica degli altri produttori di sistemi operativi. Non avendo un mercato privilegiato, non si poneva il problema del "cambiamento morbido" tra una versione e l'altra. Lo stesso manifesto di vendita era a dir poco disarmante: "vendiamo UNIX cosí com'é; senza assistenza; senza garanzie; senza addestramento;...". In pratica, tutto ció che veniva fornito era un nastro con i file sorgenti e tre manuali. La licenza era non esclusiva, e consentiva l'uso di UNIX per una particolare CPU, ogni altra CPU comportando costi addizionali. Malgrado tutto questo, UNIX contava in quegli anni circa 3000 utenti industriali di questo livello, ovvero padroni del codice sorgente.

La **AT&T** é poi passata alla fase della decisa commercializzazione. I primi risultati sono stati assai incoraggianti: le installazioni sono state valutate tra le 70.000 e le 90.000 nel 1983, e piú di 250.000 nel 1984, prevalentemente dotate di licenze "binarie", ossia non dotate di UNIX "sorgente", e si puó dire che UNIX si sta avviando a diventare uno standard per calcolatori a 16 bit.

Le caratteristiche intrinseche di UNIX da una parte, e la particolare politica di distribuzione adottata per lungo tempo dalla Bell dall'altra hanno creato lo spazio per una attività di collegamento tra il fornitore e l'utente finale. Con queste finalità si costituí nel 1977 la **Interactive** , software house che, sulla base della licenza Western, ha assunto il ruolo di fornire supporto e assistenza all'utente UNIX. La

La Interactive, oltre a potenziare gli aspetti industrialmente rilevanti di UNIX, e a renderlo più affidabile, in una versione, comunque, del tutto compatibile alla originale, ha sviluppato uno strumento di estremo interesse nell'ambito delle applicazioni di **Office Automation** : il potente text-editor ined

Versioni di Unix

- unix from AT&T: Version 7, System III, System V

- sub-licensed: Xenix, PC/IX, IS/1

- unix look-alike: Unos, Cromix, ..

- emulatori di Unix sotto altri O.S.: Eunice, IS/1

U.M.4

Le varie versioni di UNIX

UNIX V7

- Versione originale, molto diffusa, per CPU a 16 bit

- Gira su PDP/11

- Contiene migliaia di file, e i programmi in sorgente di tutti i comandi

UNIX V32

- E` per CPU a 32 bit

- Gira su Vax 11/780

- Non sfrutta appieno le capacita`del Vax, in particolare il meccanismo di impaginazione della memoria.

- La versione e`ormai obsoleta: sugli elaboratori a 32 bit si utilizzano oggi il System V o le versioni di Berkeley

L'SCCS é uno strumento molto sofisticato di gestione delle modifiche ai file. Questo insieme di programmi consente in sostanza di conservare contemporaneamente le verie versioni di un file. Notevole é anche il fatto che la parte comune delle

varie versioni viene ad essere immagazzinata, fisicamente, una sola volta. In altre parole, l'SCCS, di fronte a più versioni di un file, per prima cosa memorizza come elemento a sé stante una versione base, che può essere pensata come l' "intersezione logica" delle varie versioni, cioè la parte comune a tutte. Definisce poi un'entità, denominata **delta** , che contiene le modifiche che una versione apporta alla precedente. Ecco dunque che una particolare versione è sempre ricostruibile, a partire dalla parte comune e dai successivi **delta** fino a quello relativo alla versione richiesta. Questo metodo consente di sviluppare una struttura ad albero, la cui radice

UNIX PWB

- **PWB sta per "programmer's workbench**

- **E` per CPU a 16 bit**

- **E` pensato per offrire un ambiente completo per lo sviluppo di software per altre macchine con diverso sistema operativo (tipicamente IBM)**

- **In piu` delle versioni standard, dispone di:**

  a) SCCS (Source Code Control System) per il controllo delle modifiche dei file sorgenti

  b) REMOTE JOB ENTRY: interfaccia di tipo batch con macchine IBM

é costituita dalla versione base, e in cui i vari progressi sono percorribili per vie indipendenti a partire da essa. E' chiaro quanto uno strumento del genere risulti utile in un ambiente di sviluppo software, specie per progetti articolati e complessi.

SYSTEM III

UNIX III é la prima versione di UNIX nata dalle esigenze della industrializzazione del prodotto, e non dalla ricerca. Essa infatti é la sintesi, operata dalla Western Electric, delle tre principali versioni precedenti, ovvero l'UNIX V7, per CPU a 16 bit, l'UNIX V32, per CPU a 32 bit, e l'UNIX PWB.

1.3 IMPLEMENTAZIONI DELLA BERKELEY UNIVERSITY

L'universitá di **Berkeley** (S. Francisco, California), ha sempre rivolto particolare attenzione al fenomeno UNIX. In questo ambiente sono nate due versioni che migliorano soprattutto qualitativamente quelle precedentemente viste. Le versioni di Berkeley sono note con la sigla BSD, che sta per Berkeley Standard Distribution; al solito, sono individuate da una coppia di numeri, per poter distinguere le successive realise di una stessa versione; si parla dunque, ad esempio, di UNIX 4.2 BSD. Va notato inoltre che la Berkeley fornisce le sue versioni ed aggiunte contro il solo rimborso delle spese di spedizione a chiunque sia in possesso di una regolare licenza della Western. L'universitá di Berkeley ha sviluppato in particolare una nuova versione di UNIX V7 per calcolatori a 16 bit e di UNIX V32 per calcolatori a 32 bit. Soprattutto quest'ultima

SYSTEM III

System III = V7 + V32 + PWB

- E` per CPU a 16 o a 32 bit

- E` stato ormai trasportato su tutti i principali microprocessori a 16 bit (Intel 8086, Motorola 68000, Zilog 8000, National 16000), ed e` quindi disponibile sulla maggior parte dei "supermicro offerti sul mercato.

- Oltre a contenere interamente le precedenti versioni, migliorandone alcuni comandi, presenta varie ulteriori novita`, tra le quali ricordiamo:

  a) la gestione di un nuovo tipo di file, detto FIFO, che consente a piu` processi di comunicare tra loro, garantendo la sincronizzazione e la congruenza delle operazioni di lettura e scrittura. Questo rappresenta un potenziamento del file di tipo pipe, tipico di UNIX

  b) un processo di system accounting, che fornisce dettagliate informazioni sull'utilizzo delle risorse

  c) Una serie di compilatori per linguaggi di alto livello, come FORTRAN 77 e SNOBOL

XENIX

Lo XENIX e` una versione di UNIX prodotta dalla Microsoft, tratta dal V7, e con esso compatibile, resa adatta ai piu diffusi microprocessori a 16 bit, e ora disponibile anche in una versione compatibile con il System III.

La Microsoft ha migliorato le caratteristiche del V7 rispetto ad un ambiente di utilizzo di tipo commerciale, e lo ha arricchito con la maggior parte dei suoi popolari prodotti di software di base (compilatori, editor, spreadsheet).
Il prezzo contenuto, e la circostanza di essere stata la prima versione disponibile su tutti i microprocessori a 16 bit commercializzati, hanno fatto di XENIX la versione fino ad ora piu` diffusa.

riveste carattere di rilievo. Come si è detto, la versione originale V32 non sfrutta appieno le capacità del VAX, non gestendo la paginazione della memoria. La nuova versione, VM UNIX, fa invece ricorso al **demand paging** , e costituisce a questo punto una valida alternativa al sistema operativo nativo del VAX, il VMS. La sigla VM premessa a UNIX sta ad indicare appunto **virtual memory.** La gestione "virtuale" e non fisica della memoria consente la disponibilità di uno spazio di indirizzamento assai più vasto di quello realmente libero nella memoria fisica. Questa caratteristica si rivela particolarmente utile nelle applicazioni di tipo scientifico, grafico, e nella gestione di vasti Data Base.

I nuovi programmi e comandi delle versioni Berkeley sono raggruppabili come segue:

-Nuovi compilatori e interpreti per PASCAL, APL, LISP.

-Un potente **debugger** per PASCAL, C, FORTRAN 77.

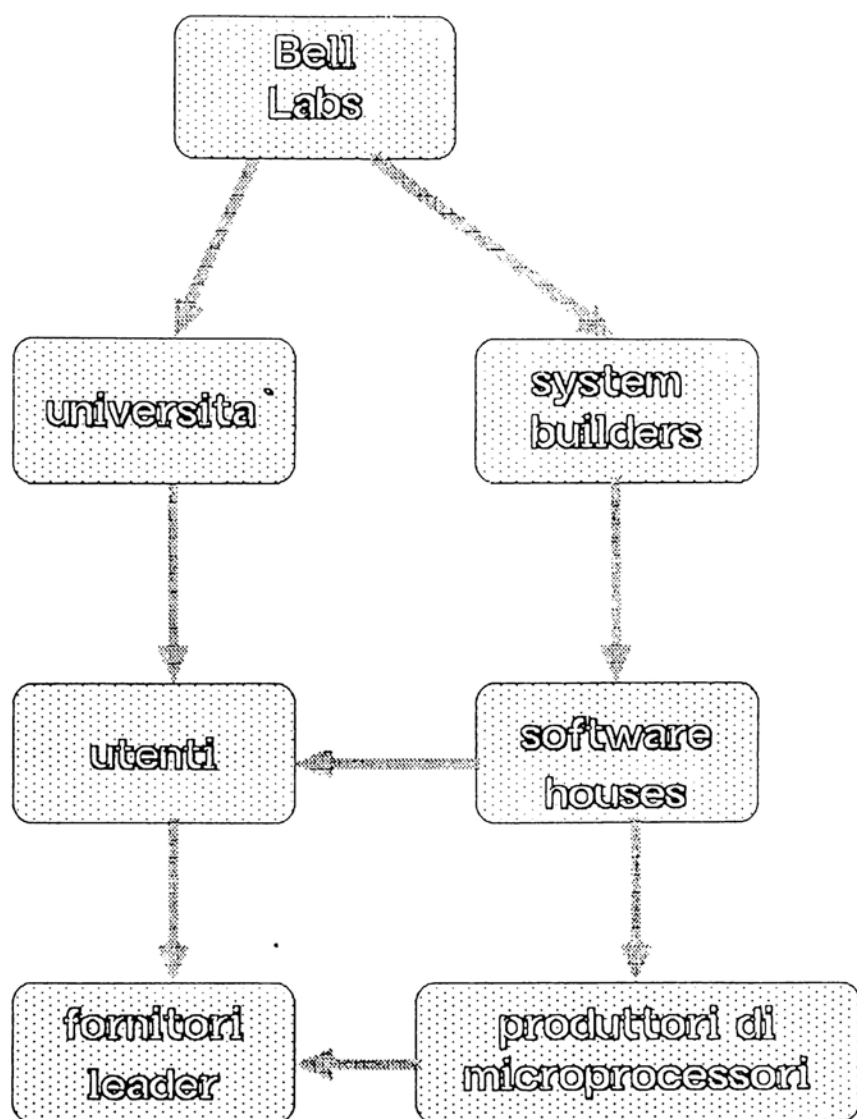
-Un nuovo EDITOR "VI" (da "video"), che consente la gestione di ogni tipo di terminale-video, intervenendo semplicemente su una tabella che lo definisce. Il VI é stato trasportato su altri calcolatori operanti sotto UNIX.

-L'introduzione dell'INGRES, un avanzato data base relazionale.

In prima approssimazione possiamo dunque definire UNIX un sistema operativo **multi user** e **time-sharing** , scritto per la maggior parte in linguaggio C , comprendente al suo interno un compilatore del linguaggio **C** , in grado quindi di autogenerarsi in modo parzialmente autonomo su una macchina a 16 o 32 bit.

Le implementazioni dell'Universita`di Berkeley

- Nuovi compilatori e interpreti per PASCAL, APL, LISP
- Un potente DEBUGGER per PASCAL, C, FORTRAN 77
- Un nuovo editor: il VI
- Un avanzato DATA BASE relazionale: l' INGRES

DIFFUSIONE DI UNIX
Bell
Labs
universita'
system
builders
utenti
software
houses
fornitori
leader
produttori di
microprocessori

Normalmente ci si riferisce a UNIX, tuttavia, pensando non soltanto al sistema operativo vero e proprio, ma anche alle centinaia di comandi che gli fanno da contorno, fondati a loro volta sul linguaggio C , e considerati ormai un tutt'uno con il sistema operativo stesso.

Un ulteriore elemento caratteristico di UNIX é lo SHELL. Esso é prima di tutto un interprete dei comandi dati direttamente dall'utente al terminale, ma risulta cosí articolato e flessibile da potere essere considerato a sua volta un linguaggio disponibile per sviluppare, come vedremo, certi tipi di applicazioni.

Nel seguito di questo capitolo descriveremo, senza scendere in eccessivi dettagli, le caratteristiche salienti di UNIX, che saranno approfondite nei capitoli successivi.

Come e` fatto UNIX

- **Parte in assembler (circa 1 kbyte)**
- **Parte in C (circa 70 kbyte)**
- **Compilatore C**
- **Comandi (programmi di uso generale scritti in C)**
- **Linguaggio Shell**

1.4 LO SHELL

Lo SHELL é prima di tutto, come si é detto, l'interprete dei comandi. Nella sua applicazione più semplice esso é dunque un programma che, ricevuto il nome di un comando da terminale, lo manda in esecuzione, e, a esecuzione finita, richiama se stesso, mostrando a video un segnale di "prompt", per ricevere ulteriori comandi. La forma generale di una chiamata dello SHELL sará dunque:

{COMANDO} A1 A2 ... An

dove {COMANDO} é il nome di uno specifico comando, e A1,...,An sono i nomi degli argomenti, separati da uno o più blank.

L'esecuzione di una procedura SHELL é strettamente legata a tre file, che lo SHELL stesso apre o crea a seconda dei casi. Essi sono lo **standard input,** lo **standard output** e lo **standard error.** Ad ognuno di essi viene attribuito un identificatore numerico, detto **file-descriptor,** che vale 0, 1 e 2 nei tre casi. Il file con descrittore 0 viene aperto in lettura, quelli a 1 e a 2 in scrittura. Nel corso dell'esecuzione dei programmi, le chiamate di input/output sono indirizzate, qualora non venga specificato esplicitamente il nome di un file, allo standard input e allo standard output. I messaggi di errore prodotti da UNIX vengono indirizzati allo standard error. L'associazione tra i precedenti file "virtuali" e i file fisici viene fatta dallo SHELL all'inizio dell'esecuzione. Normalmente (ovvero salvo esplicita dichiarazione in contrario) si ha:

```
standard input    --->   tastiera
standard output   --->   video
standard error    --->   video
```

Una delle caratteristiche dello SHELL é proprio quella di potere sostituire ai file standard qualsiasi altro file. Così, noi possiamo ad esempio decidere che lo SHELL debba leggere i dati su cui il comando dovrá operare non direttamente

dal terminale, ma da un file. Ció si ottiene con una chiamata allo SHELL del tipo:

{COMANDO} A1 A2 ... An < DATI .

Da questo momento lo SHELL esegue il comando, ma, anziché attendere i dati d'ingresso da terminale, li preleva da DATI. Inversamente, possiamo decidere che l output prodotto dallo SHELL debba costituire un file di nome qualsiasi, ad esempio RISULTATI. Impostando la chiamata:

{COMANDO} A1 A2 ... An > RISULTATI

lo SHELL creerá un file di quel nome, (qualora non esista giá) e lo sostituirá allo standard output, depositandovi ció che risulta dall'esecuzione del comando.

E' proprio nel caso di una associazione dell'output diversa da quella standard che diviene essenziale il file con descrittore a 2. Esso, a differenza di quello a 1, rimane normalmente associato al video, il che impedisce che eventuali messaggi d'errore o di sistema finiscano sul file di output all'insaputa dell'utente. Anche lo standard error puó comunque essere ridefinito, con lo stesso formato della ridefinizione dello standard output, premettendo il valore 2 al segno di maggiore:

{COMANDO} A1 A2 ... An 2> ERRORI

La piú importante conseguenza di queste caratteristiche é la possibilitá di concatenare piú programmi, in modo tale che l'output dell'uno sia assunto automaticamente come input dell'altro. Questo é un tratto essenziale di UNIX. Esso discende dalla filosofia generale del sistema, che é quella di evitare il ricorso a programmi giganteschi che pretendono di fare tutto, e di individuare invece un buon numero di funzioni relativamente semplici, molto efficienti, facilmente ricorrenti in ogni problema specifico, concatenabili a piacere.

Va infine sottolineato che lo SHELL gestisce variabili sia stringa che numeriche, salti condizionati, e quindi loop, e si

configura in conclusione come un vero e proprio linguaggio di alto livello, che vede i programmi eseguibili esistenti nel sistema come comandi elementari, quasi i propri "verbi".

1.5 PROGRAMMI E PROCESSI

Presenteremo ora, prima a livello introduttivo e successivamente in modo più approfondito, il funzionamento interno di UNIX, in particolare per quanto riguarda la gestione dei programmi lanciati dagli utenti.

Se potessimo osservare passo passo il funzionamento di un elaboratore, vedremmo che esso esegue una dopo l' altra istruzioni appartenenti sia a programmi utente che al sistema operativo in sequenze più o meno lunghe, come in figura:

```
 Prog.A   Sist.Op.   Prog.A     Sist.Op.     Prog.B   Sist.Op.
|------|-----------|---------|------------|--------|---------|...
```

Possiamo quindi desumere dalla nostra osservazione che il nostro elaboratore con sistema operativo UNIX esegue un programma per volta, sia esso lanciato dall' utente o facente parte del sistema operativo.

Anche se questa affermazione è corretta, essa è però poco esplicativa: infatti quando osserviamo una stanza in cui diversi utenti stanno operando ai terminali, ad esempio con un editor, mentre le stampanti stanno producendo lettere e tabulati, riceviamo una sensazione di simultaneità di operazione palesemente contradditoria con la spiegazione precedente.

Per risolvere questa contraddizione dobbiamo ricorrere ad un concetto più astratto di quello di programma: il concetto di "processo". Con questo termine si indica l' istanza di esecuzione di una sequenza di istruzioni su di uno specifico insieme di dati. Il processo viene inizialmente "creato", alla fine "muore" o viene "ucciso" e nel corso della sua vita può trovarsi in stati diversi. Uno stato possibile è quello già visto di esecuzione, ma inoltre un processo può trovarsi in attesa. In un sistema di elaborazione multiutente, sono presenti in ogni istante diversi processi: in generale si può affermare

che esiste un processo per ogni attività di elaborazione e/o di gestione del sistema di elaborazione che possa svolgersi "in parallelo" alle altre. In UNIX troveremo un processo (almeno) per ogni utente collegato, più diversi altri processi "di sistema" (si veda il comando **ps** che fornisce l' elenco dei processi nel sistema).

Poiché, negli elaboratori con un solo "processor", un solo processo può essere in esecuzione ad un certo istante, ogni altro processo é evidentemente quiescente, o "pronto" per l' esecuzione o "in attesa" di qualche evento, quale l' arrivo di un carattere dalla tastiera, il completamento del trasferimento di un blocco di dati da disco, o l' arrivo di un "segnale" da un altro processo con il quale esso debba sincronizzarsi. Nei periodi di attesa vengono custodite, a cura del sistema operativo, tutte le informazioni che consentono al processo di riprendere l'esecuzione laddove questa si era interrotta: il contenuto della memoria centrale utilizzata dal processo (istruzioni e dati), i valori dei registri hardware, lo stato dei file aperti, e altri dati gestiti direttamente dal sistema e non visibili all' utente fra le quali le informazioni necessarie per l' accounting. L' insieme di questi dati é denominato "immagine" del processo: in sintesi, un'immagine é lo stato corrente di un computer virtuale, la "fotografia logica" della memoria ad un certo istante dell'esecuzione. Un **processo** é l'esecuzione di una immagine. Durante l'esecuzione, l'immagine deve risiedere in memoria centrale. Dopo l'esecuzione, essa rimane in memoria centrale anche durante l'esecuzione di altri processi, fino a che un processo a prioritá maggiore non ne provoca il trasferimento su disco.

Quando viene eseguito un processo, esso viene rappresentato in memoria in tre sezioni logicamente distinte. La prima é il codice del programma in esecuzione; esso viene condiviso da qualsiasi altro processo che comporti in quel momento l'esecuzione dello stesso programma, ed é ovviamente protetto da scrittura; le altre due sono ovviamente proprie di ciascun processo: la seconda é costituita dai dati ed é libera in lettura e scrittura; la terza, infine, rappresenta lo stack, e può crescere secondo le esigenze durante l'esecuzione.

Un processo viene posto in essere quando un altro processo giá attivo esegue l'istruzione **fork**. Deve quindi esistere un processo il quale, all'accensione del sistema, dá origine alla catena di creazione di tutti gli altri. Al processo cosí sorto (che viene detto **figlio**) viene assegnata, salvo una variabile

di cui parleremo dopo, una copia esatta dell'immagine originaria del processo che lo ha creato (processo **padre**) e quindi i due processi condividono in modo completo l'accesso ai file.
A questo punto i due processi sarebbero quindi indistinguibili. A renderli riconoscibili provvede l'istruzione fork, la quale restituisce al processo padre un codice di ritorno significativo, contenente un numero univocamente attribuito al processo creato. Viceversa, il processo figlio ottiene come codice di ritorno il valore zero.

In questo modo ogni processo ha un identificatore diverso e significativo, e occupa un ben preciso posto nella gerarchia ad albero del sistema, e due esecuzioni di utenti diversi non vengono confuse ancorché siano in tutto e per tutto identiche.

Questo provoca il fatto che il padre governa il processo figlio, potendo usare il suo codice identificatore per accedere alle routines di sistema. In questo modo vengono gestiti ruoli e prioritá.

Il meccanismo descritto consente il ricorso al **pipe** , tipico di UNIX. Un **pipe** é un file di output di un processo, che viene assunto come input da un altro processo. Interessante notare che nessuno dei due processi, né il padre che passa il **pipe** , né il figlio, devono sapere che si ha a che fare con un **pipe** anziché con un file ordinario.

1.6 UNO SGUARDO AL CUORE DEL SISTEMA

Questo ed i successivi tre paragrafi sono intesi a presentare in modo più approfondito e più tecnico alcuni dei concetti giá presentati. Il lettore che non abbia alcuna dimestichezza con alcun sistema dotato di multiprogrammazione potrebbe avere qualche problema. Tuttavia, il mancato approfondimento di questa parte non pregiudica la comprensione del seguito.

Come é noto, il progressivo sviluppo del software di sistema ha fatto sí che ormai il sistema operativo si ponga come diaframma tra l'utente e tutto ció che nell'elaboratore é fisico e tangibile. Non é quindi ormai solo questione di ottimizzazione delle risorse; dalla metá degli anni '60 un sistema operativo riveste prima di tutto la funzione di creare un ambiente puramente teorico, congeniale all'utente, entro il quale sia possibile muoversi con sicurezza e conforto, demandando il maggior numero di problemi possibile alla macchina stessa. La maggior parte di queste trasformazioni da un ambiente fisico a un ambiente logico viene svolta da quello che rappresenta il cuore del sistema operativo: il **kernel** (letteralmente: nocciolo). Il kernel é il componente dei sistemi operativi che provvede alla gestione delle risorse hardware: CPU, memoria centrale, dischi, terminali, ecc.

Attraverso il complesso delle funzioni svolte, il kernel realizza un **ambiente elaborativo virtuale** più vicino alle esigenze operative degli utenti del sistema, celando le caratteristiche fisiche del sistema di elaborazione che comportano maggiori complessitá di gestione. Quale effetto delle funzioni svolte dal kernel, nei sistemi operativi **multiaccess** quale é UNIX, i programmi utente possono operare come se l' elaboratore fosse totalmente a loro disposizione (sia per quanto riguarda l' unitá di elaborazione che per la memoria centrale), senza doversi preoccupare degli altri utenti che stanno contemporaneamente utilizzando il sistema di elaborazione. Ai programmi utente, il kernel appare come un insieme di sottoprogrammi richiamabili attraverso apposite richieste, dette chiamate di sistema (system call).

Il kernel di UNIX, a differenza di quanto avviene in altri sistemi operativi, é totalmente residente in memoria centrale.

Esso è composto da un programma di circa 10.000 righe scritto nel linguaggio **C** e da circa 1.000 istruzioni nell'assembler dello specifico elaboratore su cui è installato. La sua occupazione di memoria, comprensiva delle aree dati e dei buffers, è mediamente di 90-100 kbytes per un elaboratore di tipo supermicro con 8-16 terminali.

Rispetto ad altri sistemi operativi di pari complessitá le dimensioni del kernel risultano significativamente inferiori. Le funzioni direttamente realizzate dal kernel, infatti, sono state progettate avendo in mente obiettivi di generalitá e semplicitá: questa scelta, che costituisce forse il maggior pregio di UNIX, va a volte a scapito dell'efficienza. Il pregio della semplicitá é notevolmente accresciuto nella percezione dell'utente da un'altra caratteristica notevole del kernel di UNIX, ossia la soluzione adottata per la gestione delle operazioni di input/output, cui si é giá fatto cenno. Tali operazioni vengono presentate ai programmi utenti come operazioni di lettura o scrittura su files, mascherando completamente i dettagli operativi delle diverse unitá periferiche (dischi, nastri, stampanti, terminali, ecc.).

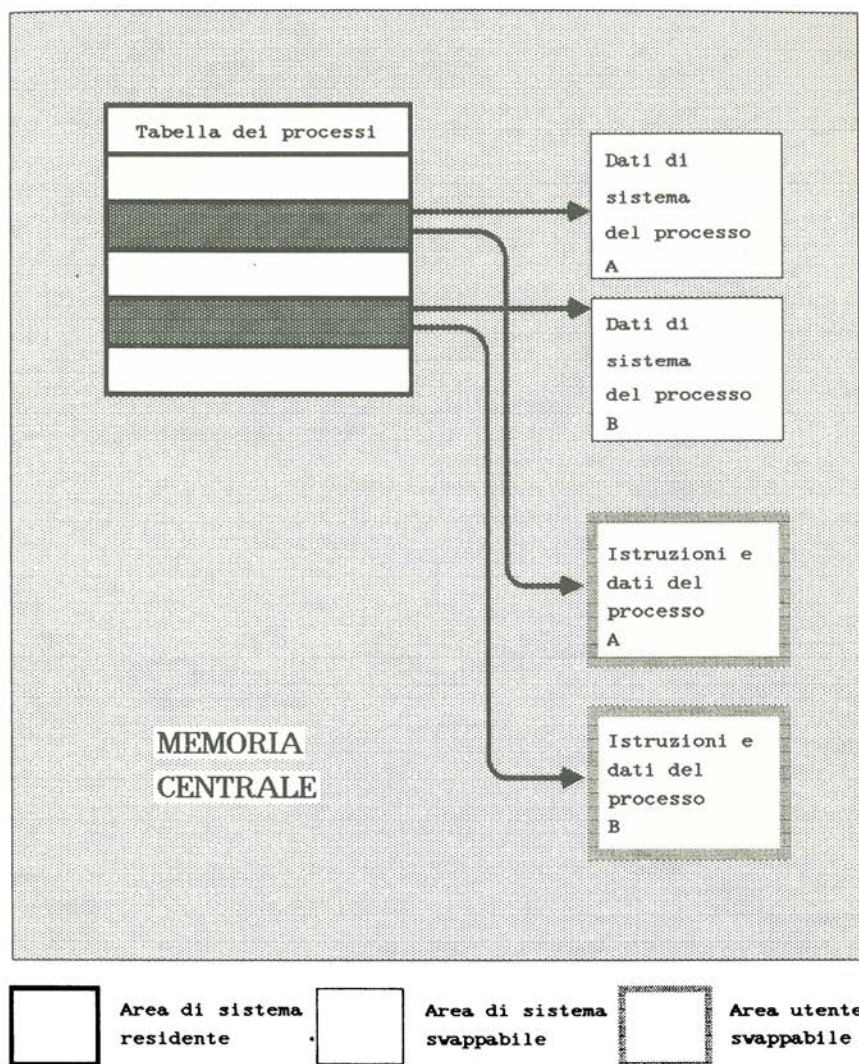
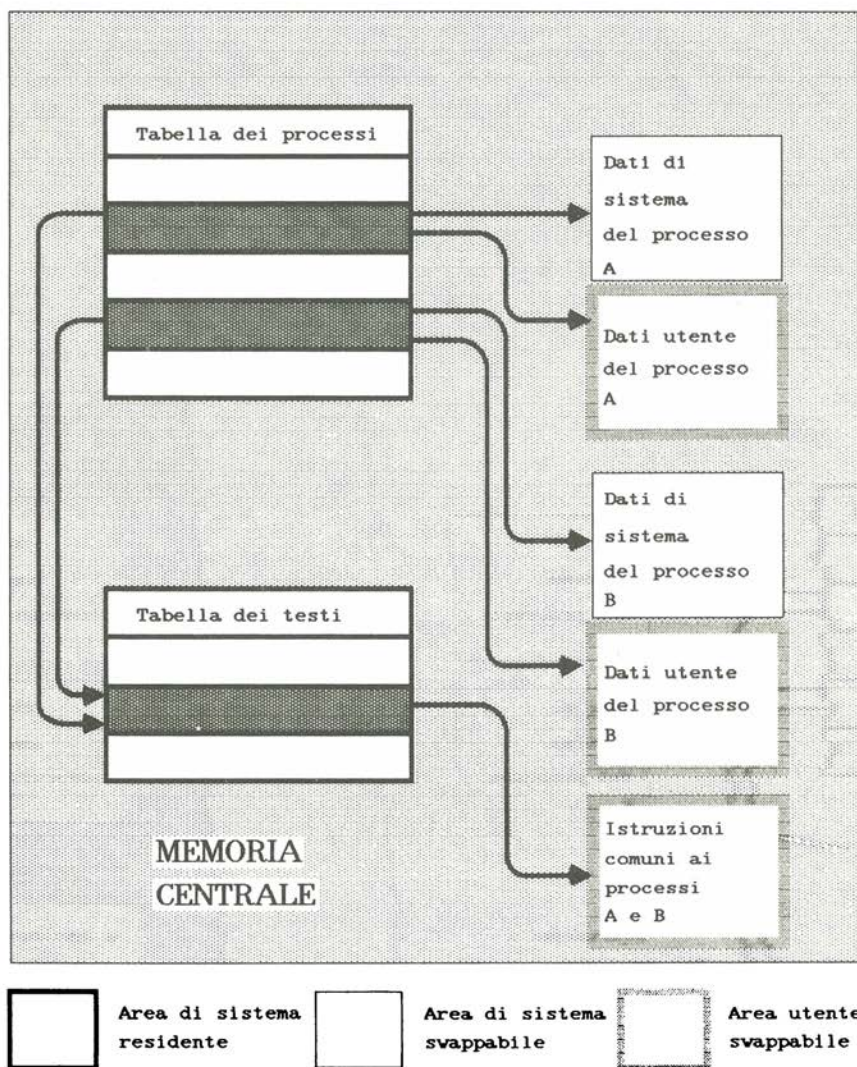
Molte funzioni che in altri sistemi operativi appartengono al kernel sono svolte in UNIX da comandi che operano come normali programmi utente, residenti su disco e non dotati di diritti speciali. I limiti del sistema operativo UNIX sono pertanto piuttosto difficili da inquadrare: il kernel rappresenta solo il 5-10 % del software che viene distribuito quando si acquista il "sistema operativo UNIX". E' inoltre possibile (e viene fatto normalmente in tutte le installazioni), aggiungere ai comandi standard forniti con la licenza UNIX programmi o procedure shell di interesse specifico che vanno, a tutti gli effetti, ad estendere quello che l'utente percepisce come "sistema operativo".

1.7 I PROCESSI

Il concetto di **processo** é di fondamentale importanza per comprendere il funzionamento dei sistemi operativi multiutente. Come abbiamo detto, con il termine "processo" si intende l'istanza di esecuzione di un programma operante su di uno specifico insieme di dati. E' quindi un concetto dinamico: i

processi nascono, attraversano diversi stati, infine muoiono. E' importante distinguere la nozione di processo da quella di programma: quando due utenti usano l'editor di sistema da terminali diversi esistono due processi utente, anche se essi eseguono lo stesso insieme di istruzioni (o programma).

Ad un certo istante (nei sistemi di elaborazione dotati di una una sola CPU), esiste al più un processo utente in stato di esecuzione. Perché un processo vada in esecuzione, come abbiamo visto, occorre che sia presente in memoria centrale l' immagine del processo, ossia l'insieme di dati, istruzioni e registri hardware che ne determinano univocamente lo stato. In memoria centrale usualmente risiedono anche immagini di altri processi non in esecuzione, ma in attesa di eventi quali la disponibilità di CPU o il completamento di operazioni di input/output.

Le strutture di dati che il kernel utilizza per gestire i processi sono rappresentate nelle due figure seguenti.

Strutture di dati per la gestione
dei processi "normali"

Strutture di dati per processi con segmento di istruzioni separato

Ogni area contigua di memoria che contiene dati o istruzioni relative ad un processo viene detta "segmento". L' immagine dei processi é costituita da due o tre segmenti, a seconda del tipo di allocazione adottato per le istruzioni. Nel primo caso viene utilizzato un solo segmento per contenere sia le istruzioni sia i dati su cui esse operano. Un altro segmento di dimensioni fisse e limitate é allocato per ciascun processo: accessibile solo dal kernel e pertanto protetto, esso contiene lo stato dei registri hardware, informazioni sui file correntemente aperti, dati di accounting, e spazio per variabili temporanee usate dal kernel.

Gli indirizzi dei segmenti sono contenuti nella **tabella dei processi** , che contiene un entry per ciascun processo. Quando un processo é creato viene allocato un entry in questa tabella, e viene liberato quando il processo termina. Ciascun entry contiene il numero del processo (un identificatore univoco progressivo), l'indirizzo dei segmenti in memoria e su disco, e informazioni utili per lo scheduling dei processi. Poiché la dimensione di questa tabella é fissata al momento della configurazione di sistema, esiste un limite massimo al numero dei processi contemporaneamente in esistenza.

UNIX consente di articolare l'immagine del processo in tre, anziché due, segmenti: istruzioni, dati utente, dati di sistema. Poiché, come abbiamo visto in precedenza, piú processi possono eseguire lo stesso insieme di istruzioni, é possibile ottenere un risparmio nell'utilizzo di memoria attraverso la condivisione (**sharing**) del testo del programma. Inoltre, poiché il segmento di istruzioni non é modificabile dal processo utente, non é necessario trascriverlo su disco quando il processo viene temporaneamente escluso dalla memoria centrale per fare posto a processi con maggiore prioritá. Si ottiene cosí sia una maggiore efficienza di utilizzo della CPU che della memoria centrale.

Una ulteriore tabella, la **tabella dei testi** , é utilizzata per contenere gli indirizzi, in memoria e su disco, dei segmenti di testo.

Ogni posizione della tabella contiene anche il numero dei processi che condividono il segmento. Quando questo numero diviene zero, la posizione nella tabella é liberata assieme alle aree allocate in memoria centrale e sul disco. Va notato che i due segmenti di dati del processo, quello utente e quello di sistema, vengono allocati in aree contigue per diminuire i

tempi di trasferimento da e verso la memoria secondaria. Mentre il segmento di sistema ha dimensioni fisse, quello utente contiene due aree di dati di dimensioni variabili. La prima, lo stack utente contenente le variabili locali dei sottoprogrammi, é automaticamente adeguata dal sistema in caso di overflow. La seconda é invece sotto il controllo del processo utente che ha a disposizione chiamate opportune per aumentarne o ridurne le dimensioni. Nei due casi il kernel cerca di trovare uno spazio sufficiente in memoria centrale, eventualmente ricopiando i due segmenti in un' altra area di memoria. Se la ricerca ha esito negativo l'esecuzione viene sospesa e l'immagine del processo temporaneamente trasferita in memoria secondaria.

1.8 LO SCHEDULING

Durante l'esecuzione di un processo, questo puó necessitare di **funzioni** offerte dal kernel quali l'accesso ad un file, l'effettuazione di operazioni di input/output, la richiesta di ulteriore memoria centrale, ecc. Nel corso della chiamata di sistema, avviene una transizione dall'ambiente di **utente** a quello di **sistema** , a seguito della quale il processo assume i privilegi del sistema operativo (possibilitá di eseguire istruzioni di input/output, accesso al segmento di sistema, ecc.). Il processo viene ora chiamato **processo di sistema.** Se il completamento della funzione richiede risorse o dati non disponibili, il processo si pone in attesa fino al verificarsi dell'evento desiderato, il quale "risveglia" il processo (o i processi) in attesa. Quando la funzione richiesta é stata assolta si ha di nuovo una transizione di stato e ritorna in essere il **processo utente.**

Se fotografiamo lo stato del sistema in un certo istante, troveremo diversi processi in attesa di eventi, altri pronti per l'esecuzione, e uno solo in possesso della CPU e quindi in stato di esecuzione. Quando il processo attivo deve a sua volta porsi in attesa di un evento, la CPU diviene libera e il kernel provvede a scegliere, fra i processi pronti per l'esecuzione, quello a piú alta prioritá.

La prioritá dei processi di sistema é sempre maggiore di quella dei processi utente. La prioritá dei processi utente é determinata dalla combinazione di due componenti: il livello

base di prioritá, che puó essere modificato dall'utente con il comando **nice** , e l'ammontare di tempo di calcolo attribuito recentemente al processo. I processi che piú hanno usufruito dell'unitá di elaborazione nell'ultimo intervallo di tempo vengono penalizzati. Ogni secondo il kernel sospende l'esecuzione e ricalcola le prioritá dei processi. L' obiettivo dell'algoritmo di scheduling é quello di privilegiare i processi interattivi e quelli che utilizzano intensamente il disco rispetto a quelli di pura elaborazione, e di distribuire equamente le risorse tra processi con uguali caratteristiche.

Finora, in questo paragrafo, abbiamo implicitamente supposto che le immagini dei processi risiedessero in memoria centrale. Normalmente, tuttavia, l'occupazione di memoria dei processi supera le dimensioni della memoria, per cui i dati di parte dei processi vengono memorizzati, secondo le necessitá, sulla memoria secondaria (disco). Il trasferimento da e verso il disco (**swapping**), viene effettuato da un processo particolare, il cui compito é quello di selezionare, fra i processi residenti in memoria, quello (o quelli) da trasferire sulla memoria secondaria e, viceversa, decidere, fra i processi temporaneamente **swapped out** , quale trasferire in memoria centrale e dove allocarlo.

Gli algoritmi usati da questo processo hanno effetti determinanti sull'efficienza complessiva del sistema:

- i processi da trasferire verso la memoria secondaria vengono scelti fra quelli in attesa di eventi "lenti" (ad esempio input da terminale), penalizzando quelli che da piú tempo risiedono in memoria centrale e quelli con larga occupazione di memoria. Per impedire il verificarsi del **trashing** , il fenomeno per cui tutto il sistema collassa a causa dell'eccessivo swapping, l'algoritmo prende in considerazione solo i processi che risiedono in memoria centrale almeno da un certo tempo.

- il trasferimento verso la memoria centrale segue approssimativamente gli stessi criteri: vengono privilegiati i processi che da piú tempo attendono la memoria centrale e quelli con dimensione minore. L'immagine del processo viene caricata nella prima area di memoria centrale che il sistema individua come sufficiente a contenerla.

1.9 CREAZIONE DI PROCESSI

Ad eccezione di due processi, **swap** e **init** , i quali vengono creati durante la fase di inizializzazione del sistema, ogni altro processo é creato attraverso la giá ricordata chiamata di sistema

```
i = fork() ;
```

Come abbiamo detto, dopo tale chiamata, a fianco del processo chiamante esiste un nuovo processo, detto **figlio** , quasi del tutto identico al processo padre. L' immagine del processo figlio é ottenuta copiando integralmente i segmenti di dati del processo padre (ad eccezione del segmento di testo puro, se esiste). L'unica differenza risiede nel valore restituito dalla fork al processo. Si ricordi che esso vale 0 per il processo figlio relativamente al padre, mentre il padre riceve come valore il numero di processo attribuito al figlio.

Un processo puó eseguire la primitiva **exec** con la quale vengono sostituiti i segmenti di testo e di dati correnti con quelli del programma specificato come parametro all'exec. Anziché generare un nuovo processo, l'exec fa sí che si passi all'esecuzione di un programma diverso; i dati precedenti vengono persi, con l'eccezione dei file aperti.

La terminazione di un processo avviene a seguito della primitiva **exit.**

I processi generati dal fork sono del tutto indipendenti dal padre: tuttavia il padre puó attendere la terminazione di uno qualsiasi dei suoi figli attraverso la primitiva **wait**

L'esempio mostra come si puó realizzare in linguaggio C l'esecuzione di un programma attraverso la creazione di un nuovo processo.

```
/* processo padre */
i = fork();
if ( i==0 ) /* processo figlio */
            exec ("nuovo programmma");
else /* processo padre */
     wait(stato);
/* continua il processo padre
   dopo la terminazione del figlio */
```

Come si puó dimostrare, l' uso combinato delle primitive fork, exec e wait consente di articolare l' elaborazione desiderata in numero arbitrario di processi, i quali possono comunicare tramite file e, come vedremo **pipe**, ovvero file temporanei con gestione first-in-first-out.

1.10 UNIX COMPARATIVAMENTE

Alcune considerazioni conclusive possono essere tratte dalla storia e dalla sommaria descrizione di UNIX che abbiamo visto. Prima di tutto, UNIX nasce da un ambiente di ricerca e non da un ambiente di produzione industriale. Nasce quindi con caratteristiche sui generis, che lo rendono un prodotto per molti aspetti diverso dal solito.

Un altro tratto specifico di UNIX dipende dalla politica di distribuzione adottata nei primi tempi dalla Bell. Dandolo quasi gratuitamente alle Universitá, la Bell ha ottenuto una notevole diffusione da un lato e un miglioramento intrinseco del prodotto dall'altro. L'impostazione generale del tipo di software, che é pensato per implementare tanti programmi specifici concatenabili, piuttosto che grossi programmi di problematica efficienza, ha fatto sí che l'uso massiccio da parte di certe universitá di UNIX abbia portato in modo naturale all'arricchimento dei programmi disponibili, che si sono fusi in un tutt'uno con il nucleo originario del sistema.

Di estrema importanza nella determinazione della fortuna di UNIX é poi la trasportabilitá. Questa é dovuta principalmente a due fattori. Per un verso, nascendo il sistema da un ambiente di ricerca, esso non é stato pensato a priori per un particolare elaboratore; per un altro verso, UNIX é scritto per la quasi totalitá in linguaggio **C** , e include a sua volta un compilatore per quel linguaggio, progettato espressamente per

facilitarne il trasporto su altri elaboratori. Questo vuol dire che per esportare UNIX da un particolare microprocessore ad un altro, tutto ciò che va fatto si riduce sostanzialmente alla riscrittura del kernel, meno di 1 kbyte, in assembler, e ad un intervento sulle routines semantiche del compilatore; da quel punto UNIX è in grado di autogenerarsi sulla nuova macchina. Una esperienza significativa in merito è stata quella dell' **Onyx** L'Onyx, produttore di microelaboratori della Sylicon Valley, è stata forse la prima a produrre industrialmente un mini basato sul microprocessore Z8000 dotato solamente del sistema operativo UNIX. Anziché investire tempo e risorse nel progetto di un ennesimo specifico sistema operativo, il produttore ha adottato lo IS/1 (versione di UNIX prodotta dalla Interactive, società di software del settore, per applicazioni commerciali), garantendosi in tal modo a priori il buon livello del risultato e una rapida realizzazione in fatto di tempo.

E' interessante seguire, anche solo sommariamente, i vari passi in cui si articola l'operazione di esportare UNIX da una macchina ad un'altra. Supponiamo di avere UNIX operante su un PDP11, e di volerlo trasportare su Onyx, tanto per attenerci ad un esempio realistico. Questo trasporto comporta (semplificando un poco) l'esecuzione delle seguenti operazioni su PDP11:

1) modifica del compilatore di C; in particolare, si lasciano invariate la parte lessicale e quella dell'analisi sintattica, e si interviene solo sulla **semantica**,ovvero sulla parte che deve generare il nuovo codice macchina. E' da notare che per questa operazione si ha nel comando **yacc** un valido supporto.

2) compilazione di UNIX sulla macchina sorgente, con il compilatore C modificato per produrre il codice per l'elaboratore target (crosscompilazione).

3) riscrittura della parte di kernel scritta in assembler per la nuova macchina.

4) compilazione del compilatore. Questo produce, su PDP11, un compilatore C, scritto nel linguaggio della nuova macchina.

A questo punto non resta che passare sulla nuova macchina, e rigenerare UNIX.

Altra caratteristica peculiare di UNIX é il **file system** ; piú ancora, é lo stesso concetto di file che sotto UNIX acquista una generalitá pressoché totale. Ogni ente sotto UNIX é un file. Il terminale, la printer, un qualsiasi programma sono file. Per un altro verso, file é sotto UNIX semplicemente un **vettore di byte.** Non esiste, per il sistema operativo, il concetto di "record". Un file é dunque logicamente organizzato in una successione di byte, e se un programma richiede una particolare struttura a record o a blocchi per i propri file, é esso stesso che deve sovrapporre tale struttura al contenuto dei file.

Va infine notato che UNIX non é un sistema **real-time**

Questa affermazione viene frequentemente fatta, ma raramente é sufficientemente motivata. Le caratteristiche che un sistema operativo deve possedere per essere considerato real-time sono derivate dall' esigenza di rispondere in modo prevedibile ad eventi esterni: il sistema deve in particolare presentare tempi di risposta garantiti e compatibili con l' applicazione. Evidentemente quanto piú stringenti sono i vincoli temporali tanto piú spinta é la natura real-time dell' applicazione. Le funzionalitá principali richieste ai sistemi real-time sono le seguenti:

- scheduling dei processi real-time sulla base di prioritá fissate

- risposta alle interruzioni in tempi definiti. Questo implica che ogni processo utente e il sistema operativo stesso siano in stato di non interrompibilitá solo per brevi periodi di tempo.

- controllo da parte dell' utente delle risorse del sistema (CPU, memoria centrale e aree di disco, locking di device di I/O e di file)

- ricchezza di interfacce di I/O per gestire la moltitudine di apparecchiature usate nelle applicazioni real-time.

- comunicazioni fra processi (semafori, messaggi, monitor)

UNIX, nelle sue versioni piú diffuse, é carente in una o piú

di queste caratteristiche. Tuttavia, sia a partire da System V che da Berkeley 4.2 BSD, si potrebbe ottenere un sistema con discrete prestazioni real-time senza snaturare le peculiarità proprie di UNIX.

Esso non é adatto quindi, nelle versioni attuali, a controlli di processi in tempo reale, o per una gestione dell'interrupt rivolta ad applicazioni di quel tipo. Né é pensato con protezioni particolarmente sofisticate. Esso si rivolge in conclusione, in modo privilegiato, ad utenti con ambienti di lavoro di massima indipendenti; tipicamente, lo sviluppo del software e l'office automation.

Schema riepilogativo

- UNIX nasce da un ambiente di RICERCA e non di produzione industriale
- E`stato arricchito di un notevole PATRIMONIO di SOFTWARE dalle UNIVERSITA`
- E`della massima TRASPORTABILITA`, poiche` contiene il COMPILATORE del linguaggio nel quale esso stesso e`scritto
- Presenta un potente FILE SYSTEM, e un concetto molto generale di FILE
- L'interprete dei comandi e`articolato sintatticamente al punto da costituire un linguaggio di alto livello
- E`poco adatto ad applicazioni che prevedano il controllo di processi in REAL TIME

CAPITOLO II

IL FILE SYSTEM

2.1 LA FILOSOFIA DI UNIX

Per comprendere lo spirito del **file system** operante sotto UNIX bisogna por mente al modo nel quale esso é sorto. Come si é detto nella presentazione generale, l'antenato diretto di UNIX é stato il **multics.** La filosofia del **multics** puó essere sintetizzata nel seguente motto:

costruire sul lavoro di altri

Questo significa evidentemente prospettare l'adozione piú larga possibile della programmazione **modulare.** ovvero di quella forma di approccio che tende, dato un problema, a scomporlo in tanti sottoproblemi, la cui soluzione, resa piú semplice dall'ambito limitato, sia poi concatenabile a tutte le altre, a costituire la soluzione finale.

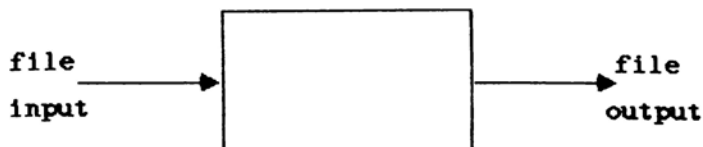
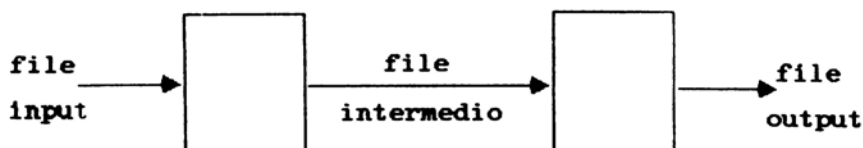
In questo ordine di idee ci si é posti nell'ottica di utilizzare, quale base per la modularitá, una classe di programmi denominati "filtri". Un filtro é un programma che riceve il suo input da un file e produce il suo output su di un file. Molti filtri operano semplici trasformazioni sui dati (modificare un carattere in un altro, ordinare un file, contare le linee, le parole, ecc.). Esistono anche filtri notevolmente complessi (si veda "grep"), che tuttavia condividono con quelli piú semplici la caratteristica di svolgere un ristretto numero di funzioni. In effetti, nel manuale UNIX la descrizione di un comando é normalmente contenuta in una o due pagine.

```
                 ------------
file            :            :            file
input  ------->:            :------>    output
                :            :
                 ------------
```

Costruire sul lavoro degli altri

A - Tutto e`un file

B - Redirezione dell'INPUT/OUTPUT

C - L'OUTPUT di un programma puo`essere assunto direttamente come INPUT di un altro (PIPE)

L' obiettivo è quello di costruire, in molti casi senza scrivere neppure una riga di software, procedure di elaborazione basate sulla concatenazione di più filtri:

Ad esempio, la produzione della lista di un programma, con le istruzioni numerate progressivamente e corredata delle opportune intestazioni in ogni pagina, per cui in UNIX non esiste un comando apposito, può essere realizzata molto semplicemente con la concatenazione dei comandi **nl** e **pr**.

Un importante elemento per la costruzione di procedure basate su filtri è il **pipe**.

Il pipe è un' area di comunicazione in memoria centrale fra due filtri (operanti come processi distinti) il primo dei quali scrive su di esso (quindi simula il file di output) mentre il secondo lo legge. Nella figura precedente, quindi, il file intermedio può essere sostituito da un pipe ottenendo un incremento di efficienza (in quanto non sono più necessari trasferimenti di dati su disco) e di semplicità della procedura non essendo più necessaria la creazione e la cancellazione del file temporaneo.

Poiché l'input e l'output dei filtri sono file, la necessità

subito insorta é stata quella di ampliare il più possibile il concetto di file, per giungere ad un grado di generalitá tale che sotto di esso potessero rientrare tutti i file a cui qualsiasi filtro fosse interessato. L'esito di questo modo di pensare puó essere riassunto nei tre fondamentali punti seguenti:

a) tutto é un file - ossia tutte le strutture di dati visibili dall'utente sono da esso trattabili in modo omogeneo ai normali file allocati su disco. Inoltre i file sono trattati, da parte del sistema operativo, in modo del tutto destrutturato: un file é, per UNIX, solo un vettore composto da 0 o più byte.

b) poiché tutto é un file, l'input e l'output di un programma possono essere, indifferentemente, file nel senso tradizionale del termine o device, come terminali o stampanti (redirection dell'I/O).

c) l'output di un programma puó essere direttamente assunto come input di un altro programma (attraverso i **pipe** in modo che i programmi siano facilmente concatenabili.

Esistono sotto UNIX tre tipi fondamentali di file:

a - i file ordinari

b - i file speciali

c - i directory

Un file ordinario é, come si é detto, una sequenza ordinata di byte. Una linea é separata dalla successiva da un carattere speciale, che dunque si presenta, a sua volta, come un byte.

In conclusione, la struttura dei file é controllata dai programmi che li usano, e non dal sistema operativo.

Un directory é un file che assicura il collegamento tra i nomi dei file e i loro contenuti. Di massima, un directory puó essere trattato come un file ordinario (ad esempio puó esserene letto il contenuto con il comando **od.** , ma non puó essere scritto altro che da programmi dotati delle opportune autorizzazioni. All'utente, il proprio directory si presenta come un elenco recante i nomi dei propri file. Tra questi ultimi, possono essere posti ulteriori file di tipo directory, ovvero sub-directory del directory principale. File e directory si presentano dunque nella classica struttura ad albero.

Tipici dell'UNIX sono poi i file speciali. Essi sono quelli associati alle unitá di input/output (tastiere, video, stampanti, nastri, ecc.). Un file speciale é letto e scritto dai programmi scritti dall' utente come un file ordinario. Ogni file speciale é associato ad un particolare device e ad uno specifico driver di input/ output. Tutti i driver di i/o, in UNIX, sono stati realizzati in modo da presentare la stessa interfaccia e cioé le stesse modalitá di uso, naturalmente adattando il comportamento effettivo alle caratteristiche fisiche del device (ad esempio il posizionamento all' inizio del file é trattato dal driver che gestisce i file su disco modificando il puntatore corrente mentre il driver che gestisce il nastro emette un comando di rewind). Naturalmente, opportune protezioni evitano accessi indiscriminati ai dischi di memoria attivi, i quali sono a loro volta visti come file. Questo modo di concepire le unitá di I/O presenta il vantaggio di rendere l'accesso ad esse il piú possibile simile all'accesso a un file ordinario. Inoltre rende facile ottenere che un programma prelevi il suo input o rilasci il suo output, indifferentemente, da o su un file o una unitá specifica. Infine lo stesso sistema di protezioni governa sia le unitá di I/O sia i file ordinari.

2.2 LA STRUTTURA AD ALBERO

Abbiamo detto che un directory puó essere pensato come un elenco di file. Tra i suoi file, un directory puó ovviamente contenere altri directory. Questa situazione realizza dunque una struttura gerarchica, ossia, in termini di teoria dei grafi, un **albero.**

La struttura in cui sono organizzati i file (normali o directory che siano) ha una radice unica, base del sistema, che prende il nome di **root-directory.**

Immediatamente sotto a root si trovano altri directory che possiamo considerare "di sistema":

```
bin        etc        tmp        usr      lib
dev        priv       u
```

Questi directory contengono di massima comandi, librerie e la definizione dei device presenti; la configurazione di questa parte della struttura é dunque standard, e su di essa deve intervenire, con modalitá precise, chi gestisce l'installazione, ossia il sistemista. Un discorso a parte merita invece il directory **u.**

In esso sono definiti gli utenti autorizzati all'utilizzo del sistema. Quando il sistemista definisce un utente, egli non fa altro che creare un nuovo sotto-directory di **u.** Di solito, il nome scelto per tale directory é precisamente il nome dell'utente. Questo fatto risulta comodo perché tale nuovo directory diviene il **login directory** dell'utente, ossia quel directory in cui l'utente si trova quando esegue la procedura per registrarsi nel sistema.

Vediamo la cosa con un esempio. Supponiamo di avere definito, in **u.** gli utenti **maurizio paolo giulia angela.** La nostra struttura sará dunque

A livello 1 troviamo **root**. il cui nome é la barra /.

a livello 2, ossia direttamente sotto root, troviamo una serie di directory di sistema, tra i quali **u**. che contiene i directory corrispondenti agli utenti ammessi all'uso del sistema, i quali sono quindi di livello 3.

Un utente, diciamo **maurizio**. che voglia utilizzare il computer, dovrá per prima cosa eseguire la procedura di login. Il sistema chiede all'utente il nome, e poi la password, se é stata definita. Quando il nome viene fornito, viene eseguito il controllo che esso esista in **u**. ossia che corrisponda a un utente autorizzato.

Un utente entra quindi nel sistema ad un determinato livello della struttura gerarchica, ossia nel suo login directory. Da questo punto della struttura verso il basso, tutta l'organizzazione dei directory é completamente libera, ovvero viene scelta dall'utente.

Supponiamo dunque di eseguire un login: il sistema propone la domanda

login:

a cui rispondiamo

maurizio

Poiché non é stata definita una password, il sistema non la chiede; un prompt (**$**. ci avvisa che la procedura di login é stata completata, e che il sistema é pronto ad eseguire i nostri comandi.

Diamo ora il comando **pwd.** In risposta ad esso, il sistema fonisce il **path-name.** o "nome-tracciato" del directory che attualmente ci ospita:

/u/maurizio

Il primo segno della risposta che abbiamo ottenuto rappresenta, come abbiamo visto, il nome della radice del nostro albero; lo stesso segno / compare di solito altre volte nei nomi-tracciato, ma semplicemente con le funzioni di separatore. Dopo il nome di root, abbiamo **u.**

quindi un separatore, e infine **maurizio.** Il sistema quindi ci ha fornito il nome dell'attuale directory di lavoro e la sua precisa posizione all'interno della struttura ad albero: ci ha informato infatti che **maurizio** sta sotto a **u.** e che questo sta sotto a /. Il significato dei nomi-tracciato é proprio questo, quello di individuare univocamente una posizione nella struttura ad albero.

Possiamo ora strutturare il nostro ambiente di lavoro a piacimento, ovvero implementare verso il basso la struttura ad albero. Nel directory **maurizio** creiamo allora liberamente dei sotto-directory. Il comando che consente la creazione di un directory é

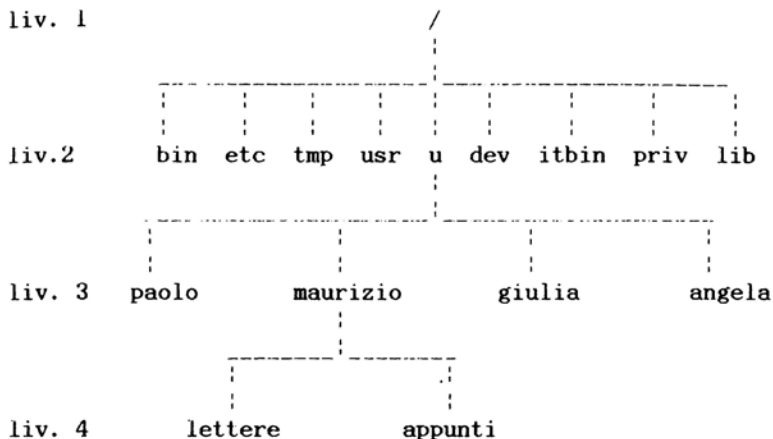
mkdir

Potremmo, ad esempio, creare un ulteriore directory di nome lettere.

mkdir lettere

Allo stesso modo possiamo creare **appunti.**

mkdir appunti.

La nostra struttura é allora, attualmente, la seguente:

L'inverso del comando **mkdir.** ossia quello per cancellare un directory, è **rmdir** (il primo abbrevia "make directory", il secondo "remove directory"). E' da notare che un directory si può cancellare solo se non contiene file.

Il comando per passare da un directory a un altro, ovvero per muoversi all'interno della struttura, è

cd

dalle inziali di "change directory", cambia directory. Se vogliamo entrare nel directory **appunti** che abbiamo ora creato, dovremo dare il comando

```
cd appunti
```

Se ora diamo di nuovo il comando **pwd.** ovvero chiediamo al sistema qual è l'attuale directory di lavoro, otterremo in risposta:

```
/u/maurizio/appunti
```

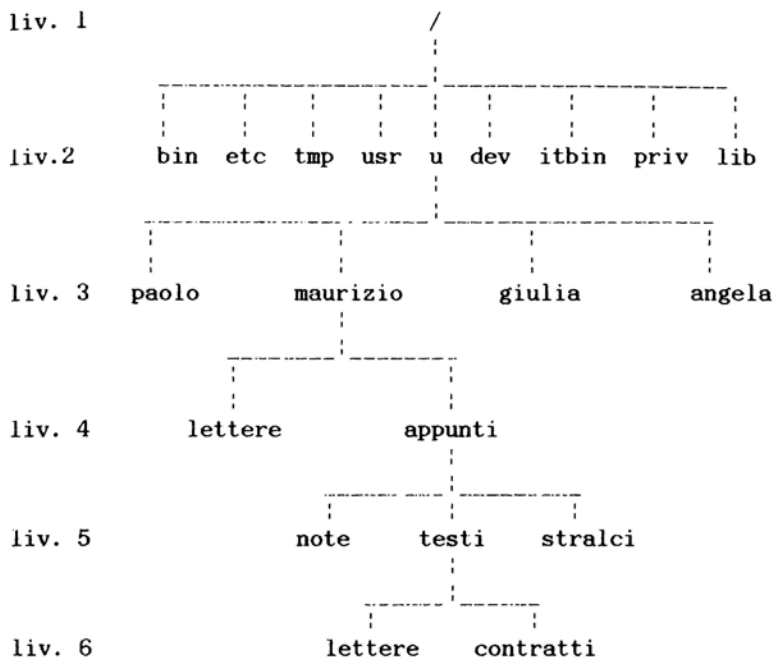
ovvero il path-name completo del directory in cui ci siamo spostati.

Il comando cd dato senza argomenti fa sí che si ritorni al proprio login directory. Cosí, se ora diamo il comando

cd

ci troveremo di nuovo nel directory /u/maurizio.

Questo metodo di identificare i directory, e piú in generale i file, attraverso nomi tracciato che ne identifichino anche la posizione consente la massima libertá nell'attribuire i nomi, poiché di fatto si é tutelati da possibili omonimie. Ad esempio, anche l'utente **paolo** puó creare un directory di nome appunti. che tuttavia verrá identificato dal sistema con il nome completo **/u/paolo/appunti** anziché con **/u/maurizio/appunti**. Il nome tracciato di ogni ente é dunque unico, e questo fatto elimina qualsiasi confusione. Tuttavia sarebbe scomodo dovere fare riferimento ai file fornendo sempre il nome tracciato completo. La convenzione adottata da UNIX é allora quella che é possibile riferirsi a tutto ció che sta sotto all'attuale directory di lavoro fornendo il nome tracciato **relativo** e non quello assoluto. Il nome tracciato relativo sará quello che il file avrebbe se il nostro directory attuale fosse la radice dell'albero. Un nome tracciato viene identificato come relativo anziché come assoluto per il fatto che esso non comincia con il carattere /.

Complichiamo ulteriormente la nostra struttura, per potere fare qualche esempio significativo:

I nomi tracciato relativi dei due directory di **maurizio**

saranno semplicemente **lettere** e **appunti**. i tre sottodirectory di **appunti** saranno rispettivamente

appunti/note

appunti/testi

appunti/stralci,

mentre i sottodirectory di **testi** saranno

appunti/testi/lettere

e

appunti/testi/contratti.

In questo modo si continua a eliminare qualsiasi ambiguitá, ma si evita di dover fornire il nome tracciato completo; in particolare, ci si puó riferire con il nome semplice a tutti quegli enti (file normali o directory) che appartengono all'attuale directory di lavoro.

E' da notare che, essendo i directory elenchi di file, essi non conterranno ovviamente, in generale, soltanto altri directory, ma anche file che non sono directory. Dal punto di vista della struttura ad albero l'unica differenza consiste nel fatto che naturalmente un file che non sia un directory non puo ', per definizione, contenere sottodirectory, ed é quindi sempre un punto terminale, una foglia dell'albero. Ma tutto quanto detto sul modo di riferirsi agli enti facendo ricorso ai nomi tracciato vale indifferentemente per i file come per i directory.

In generale abbiamo visto quindi che dobbiamo fornire il nome tracciato completo solo quando vogliamo che la ricerca venga fatta da parte del sistema a partire da /. altrimenti, ossia in tutti quei casi in cui il file é raggiungibile a partire dal nostro directory attuale, é sufficiente fare ricorso al nome relativo. Un buon metodo che si puó consigliare ai principianti per fare pratica, é quello, quando si deve dare un comando che si riferisca a dei file, di ritornare rapidamente al proprio login directory con il comando **cd.** e fare poi ricorso ai nomi relativi al nostro login directory, che normalmente ci sono piú familiari.

Una ulteriore semplificazione introdotta per rendere piú veloce possibile il riferimento ai file, evitando il ricorso ai nomi tracciato completi, é quella che il sistema interpreta i due punti su di una stessa riga come il nome del directory padre di quello attuale, e un solo punto come il nome del directory corrente. Se ad esempio ci troviamo nel directory maurizio. e diamo il comando

```
cd ..
```

passeremo al directory **u.** Noi possiamo allora riferirci ad un file che non é raggiungibile a partire dal nostro directory corrente, ma che lo é dal directory padre, fornendo il nome tracciato relativo a quest'ultimo a cui vengono premessi i due punti. Se ad esempio siamo nel directory **paolo.** e vogliamo passare al directory

/u/maurizio/appunti/testi,

potremo dare il comando

cd ../maurizio/appunti/testi,

evitando di dovere prima risalire l'albero e poi dare il comando. La presenza dei due punti nella prima parte dell'argomento provoca infatti la risalita di un livello; la seconda parte dell'argomento é il nome tracciato del directory

testi relativamente ad **u**. Allo stesso modo, possiamo riferirci a due livelli sopra di noi, ricorrendo a

../..

e cosí via. Queste possibilitá risultano particolarmente utili se usate congiuntamente ai comandi di manipolazione dei file, come quelli per copiare un file (**cp.** , o per spostarlo da un directory ad un altro (**mv.**

Dopo avere visto da un punto di vista teorico quale sia la struttura sia esterna che interna dei file e dei directory, prendiamo in considerazione i comandi di uso piú frequente sui file. Questo modo di procedere consentirá anche al principiante di compiere le operazioni piú consuete, in modo da poter cominciare ad operare in ambiente UNIX. Il lettore esperto puó qui scorrere molto velocemente le pagine, poiché non si danno nel seguito immediato contenuti particolarmente complessi; tuttavia al neofita (di UNIX o addirittura di informatica) sará utile, avendone ovviamente la possibilitá, eseguire effettivamente gli esercizi e le prove che via via vengono proposti.

2.3 L'ELENCO DEI FILE DI UN DIRECTORY

Il comando **ls** consente di fare l'elenco di un directory. Esso é quindi uno di quelli di uso più frequente. Ogni volta che si vuole verificare l'effettiva esistenza di un determinato file in un directory, o il suo nome esatto, si puó "listare" il contenuto del directory in questione con tale comando.

Con l'opzione -l si otterranno, anziché i soli nomi dei file, alcune rilevanti informazioni supplementari. Di seguito é offerto un esempio di un elenco del directory /u/maurizio/testi ottenuto con il comando **ls -l**.

```
-rw-r--r--    1 root  2317  Aug  8 11:41  Aug  8 11:59 note
-rw-rw-r--    1 root     0  Aug  8 11:41  Aug  8 11:41 stralci
drwxrwxr-x    2 root    32  Aug  8 11:37  Aug  8 11:37 testi
total 7 blocks
```

La prima colonna comincia per **d** o per - a seconda che loggetto in questione sia un directory o un file. Gli altri segni della stessa colonna specificano le protezioni del file o del directory, di cui abbiamo giá parlato in dettaglio.

La seconda colonna indica, per i file ordinari, il numero di link attivi, ossia il numero di nomi diversi sotto i quali é individuato il file. Nel caso di un file di tipo directory, indica il numero dei nomi di altri directory riconosciuti da esso, ossia il numero di nomi di directory che sono sottoposti a quel directory, aumentato dei due nomi speciali .. e ., l'uno per il directory padre e l'altro per il directory stesso.

La terza colonna specifica il nome dell'utente che é "proprietario" del file o del directory.

La quarta colonna specifica la lunghezza del file o del directory espressa in byte.

La quinta colonna in un primo tempo specifica la data e l'ora

della creazione del file o directory corrispondente. Poiché, come vedremo, gli editor creano una nuova copia del file su cui operano, essa corrisponde alla data e ora dell'ultima sessione di editing.

La sesta colonna indica la data e l'ora dell'ultimo accesso al file o directory. Il sistema considera un "accesso" non soltanto una sessione di editing, ma anche la semplice visione, o la stampa, o il produrre le bozze.

La settima colonna, infine, specifica il nome del file o directory.

L'ultima riga indica il numero complessivo dei blocchi del directory di cui si é fatto l'elenco.

Chi usa una versione di UNIX dotata dell'editor **ined.** come ad esempio quella del PC IBM, accanto ai file appositamente creati con un comando, facendo l'elenco del proprio directory, se ne troverá altri creati dal sistema, facilmente riconoscibili perché hanno lo stesso nome del file originale con l'aggiunta di Tali file sono creati automaticamente ad ogni sessione di editing, e rappresentano il salvataggio dell'ultima versione del file prima che si aprisse l'attuale fase di lavoro. Questo fatto é molto utile, perché consente l'eventuale recupero di situazioni precedenti in caso di errore. Il sistema accetta nomi di file e directory di al massimo 14 caratteri. E' peró consigliabile attribuire ai file nomi di al piú 10 caratteri, perché altrimenti quando il sistema crea i rispettivi file .bak, il suffisso aggiunto viene a cancellare una parte del nome. Puó cosí essere poco agevole riconoscere il nome che si é attribuito, e puó anche succedere che due file aventi due nomi simili, che si differenziano soltanto dall'undicesimo carattere in poi, siano reduplicati dal sistema in file .bak dello stesso nome. Questo ovviamente crea confusione quando si ha bisogno di servirsi del file copia.

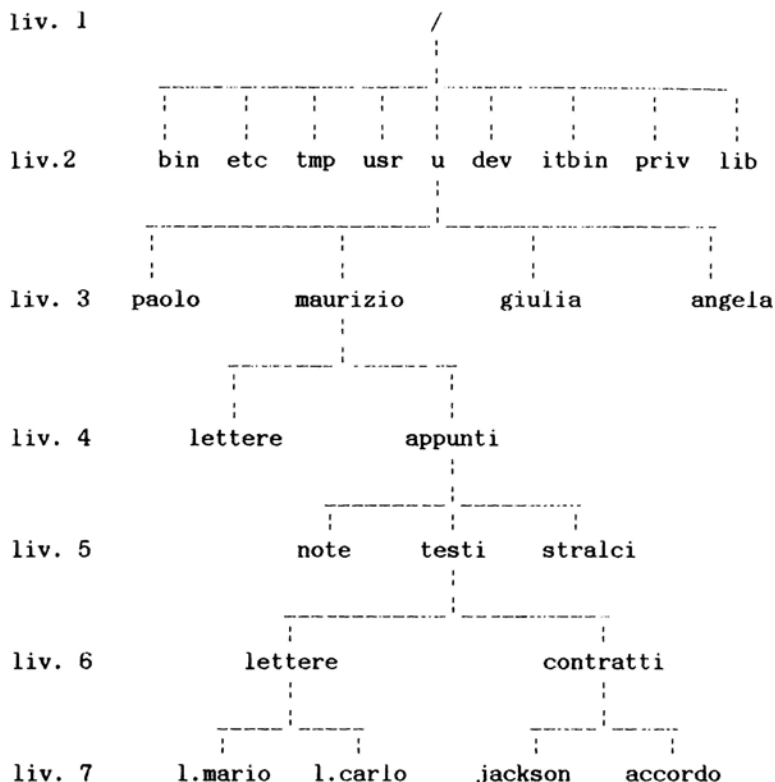
2.4 COPIARE I FILE

Un altro comando di uso assai frequente é **cp.** che consente di copiare un file. Il suo formato é

cp {nome del vecchio file} {nome del nuovo}

E' necessario lasciare uno spazio tra il nome del vecchio e quello del nuovo file. Attenzione: come sotto ogni sistema operativo, se si copia un file su un altro che giá esiste, si cancella il contenuto di quest'ultimo. UNIX non chiede conferme in questo caso, e bisogna quindi fare attenzione prima di dare il comando.

A proposito del comando **cp.** valgono tutte le considerazioni fatte in precedenza sul modo di riferirsi ai file. Se ad esempio si vuole copiare un file che non é raggiungibile cercando verso il basso a partire dall'attuale directory di lavoro, bisognerá fornire il suo nome tracciato completo.

Quando si copia nell'ambito del proprio directory attuale, é come al solito sufficiente fornire il nome relativo dei file. Per copiare un file da un directory ad un altro, bisogna ricorrere al nome tracciato completo o relativo al directory di lavoro quando non si sta lavorando nel proprio login directory. Per chiarezza ricorriamo ad un esempio. Sia la seguente la nostra attuale struttura di directory:

Gli oggetti che compaiono nella struttura sono tutti directory, eccetto quelli di livello 7, che sono file ordinari.

Supponiamo di voler copiare il file **accordo** nel directory

lettere (quello di livello 4), e di star lavorando nel directory **maurizio.** Il comando corretto é allora

cp appunti/testi/contratti/accordo lettere

Se avessimo voluto cambiare nome al file destinazione, e chiamarlo, ad esempio, **contratto.** avremmo dato:

cp appunti/testi/contratti/accordo lettere/contratto.

Nel caso considerato entrambi i directory coinvolti, sia quello di partenza che quello di destinazione, sono raggiungibili a partire dal proprio directory di lavoro; basta dunque fornire i nomi relativi dei file in questione.

Supponiamo invece di stare lavorando nel directory **contratti.**

Vogliamo compiere la stessa operazione di prima. In comando corretto é allora

cp accordo /u/maurizio/lettere,

oppure

cp accordo /u/maurizio/lettere/contratto

se vogliamo cambiare il nome del file copia da **accordo** in contratto. In questo caso siamo costretti a far ricorso al nome tracciato completo per il file destinazione. Questo perché il directory **lettere** non é raggiungibile a partire dall'attuale directory di lavoro, e dobbiamo quindi far sí che la ricerca da parte del sistema abbia inizio dal root directory /.

Consideriamo infine l'esempio inverso: noi stiamo lavorando in

lettere. e vogliamo compiere la solita operazione. Daremo allora il comando

cp /u/maurizio/appunti/testi/contratti/accordo accordo

o, nel caso vogliamo cambiare nome,

cp /u/maurizio/appunti/testi/contratti/accordo contratto.

Qui, inversamente a prima, dobbiamo fare ricorso al nome tracciato completo per riferirci al file sorgente anziché al file destinazione. Si ricordi che il segmento iniziale, **/u/maurizio.** puó essere sostituito con i due punti orizzontali.

2.5 CAMBIARE NOME AI FILE

Ci siamo soffermati sul comando **cp** per rendere più chiare possibili le modalitá di riferimento ai file nell'ambito della struttura del file system. Tali modalitá rimangono del tutto invariate nel caso di qualsiasi altro comando che presupponga dei file ad argomento. Cosí ora, molto più speditamente, dando per scontato tale fatto, ricordiamo qualche altro comando di uso assai frequente.

Il comando **mv** viene usato con la stessa sintassi di **cp.** Esso consente di spostare un file da un punto all'altro della struttura, o di cambiargli il nome, o di fare entrambe le cose contemporaneamente. In altre parole, il suo funzionamento é identico a quello del comando **cp.** con la essenziale differenza che esso non duplica il file di partenza, ma lo sopprime nel creare quello destinazione. Come lo si possa usare per cambiare il nome di un file, é semplice da capire. Supponiamo di essere nel directory **contratti.** e di volere cambiare nome ad accordo. chiamandolo **bozza.** basterá dare il comando

mv accordo bozza.

2.6 LA CONCATENAZIONE E LA CANCELLAZIONE

Un altro comando estremamente utile, su cui ci soffermeremo più in dettaglio nel capitolo sullo SHELL, é **cat.**

Esso consente, tra l'altro, la concatenazione di più file in un unico file destinazione. La sua sintassi é

cat filel file2 > file3

Dopo l'esecuzione del comando, il contenuto di file3 sará quello di filel e file2 l'uno dopo l'altro. L'eventuale precedente contenuto di file3 va distrutto. Per appendere in coda a file3, conservandone il contenuto iniziale, bisogna semplicemente raddoppiare il segno di maggiore:

cat filel file2 >> file3.

Per cancellare un file, sono disponibili i comandi **rm** e **del**. La differenza sta nel fatto che il primo non chiede conferma, e il secondo sì. Attenzione: UNIX interpreta come una conferma l'eventuale risposta di ritorno carrello; per evitare la cancellazione dopo avere dato il comando **del** alla domanda

```
delete file1? (y)
```

bisogna rispondere esplicitamente di no.

CAPITOLO III

LA STRUTTURA INTERNA DEI FILE

3.1 STRUTTURA INTERNA DEI FILE

Un directory contiene un elenco di nomi di file. L' elenco dei file contenuti in un directory può essere ottenuto, come abbiamo visto nel capitolo precedente, attraverso il comando ls. Combinando opportunamente le numerose opzioni offerte da ls la lista dei file può essere integrata da numerose altre informazioni.

La struttura interna dei directory è tuttavia molto semplice: essi catalogano i file attraverso due soli elementi:

Il nome del file

Un puntatore al file, detto I-NUMBER (da index number).

Possiamo verificare questa affermazione esaminando direttamente il contenuto fisico di un qualsiasi directory, accedendo ad esso come ad un normale file. Per questo si può utilizzare il comando **od** (octal dump). Supponendo, ad esempio, di essere posizionati sul directory /u/maurizio/appunti, per vederne il contenuto "fisico" si può fornire il comando:

```
od -bc .
```

Sul terminale apparirá la seguente mappa:

```
0000000  \t 256   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
        011 256 056 000 000 000 000 000 000 000 000 000 000 000 000 000
0000020 007   j   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
        007 152 056 056 000 000 000 000 000 000 000 000 000 000 000 000
0000040  \t   W   t   e   s   t   i  \0  \0  \0  \0  \0  \0  \0  \0  \0
        011 127 164 145 163 164 151 000 000 000 000 000 000 000 000 000
0000060  \b 272   n   o   t   e  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
        010 272 156 157 164 145 000 000 000 000 000 000 000 000 000 000
0000100  \t 255   s   t   r   a   l   c   i  \0  \0  \0  \0  \0  \0  \0
        011 255 163 164 162 141 154 143 151 000 000 000 000 000 000 000
0000120  \0  \0   v   e   r   s   i  \0  \0  \0  \0  \0  \0  \0  \0  \0
        000 000 166 145 162 163 151 000 000 000 000 000 000 000 000 000
0000140
```

Ogni riga del dump é formata da 16 caratteri, i primi due dei quali contengono un numero intero, lo I-NUMBER, mentre i restanti 14 sono a disposizione per il nome del file. Ogni directory contiene sempre due entry per i file . e .. corrispondenti rispettivamente al directory stesso e al directory padre. Va osservato che nel dump é contenuto anche il file **versi**

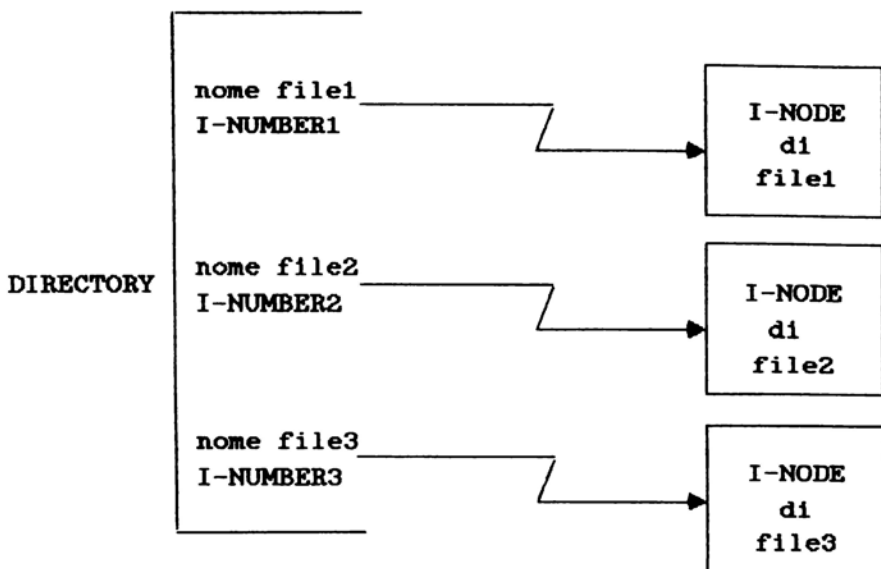
che non compariva a seguito del comando **ls.** Questo file é in effetti cancellato (lo si deduce dall' I-NUMBER nullo), ma la entry corrispondente non viene eliminata. Il sistema operativo tuttavia provvede a riutilizzare le entry in occasione della creazione di nuovi file.

Lo I-NUMBER consente l'accesso a una tavola di sistema, detta

I-LIST .

La I-LIST non é altro che un elenco, quindi, che contiene le informazioni salienti relative ad ogni file presente nel directory. L'insieme delle informazioni relative ad un determinato file prende il nome di

I-NODE .

Il seguente schema servirá a individuare la situazione.

Un I-NODE, che occupa 64 byte, contiene tutte le informazioni essenziali relative al suo file, e precisamente:

1- L'utente e il gruppo del suo proprietario

2- I suoi bit di protezione

3- Gli indirizzi che consentono di accedere (nel seguito si vedrá come) al contenuto del file

4- La dimensione del file

5- La data di creazione, di ultimo accesso, di ultima modificazione

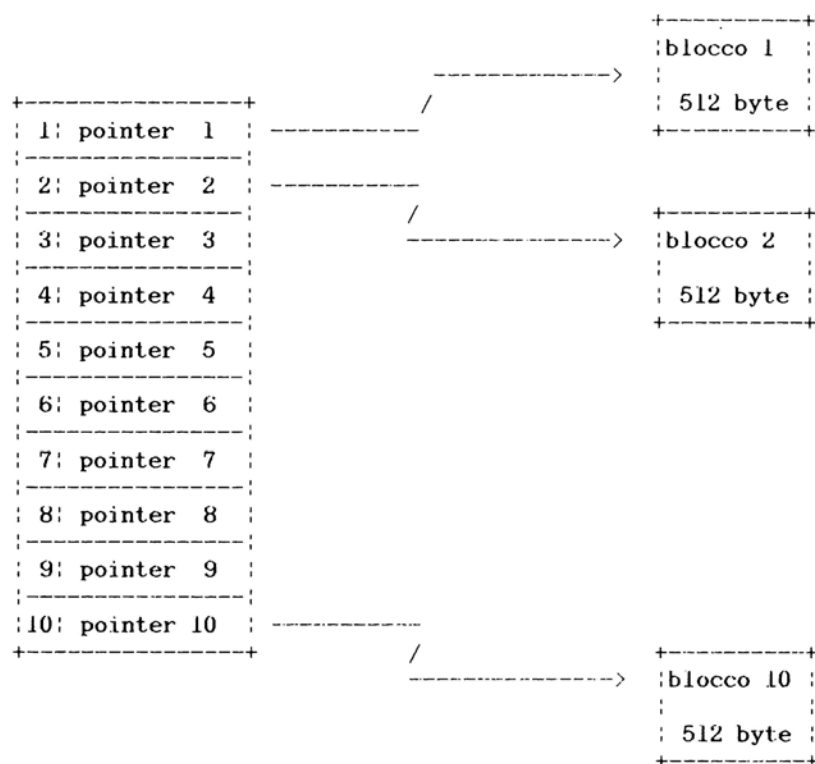
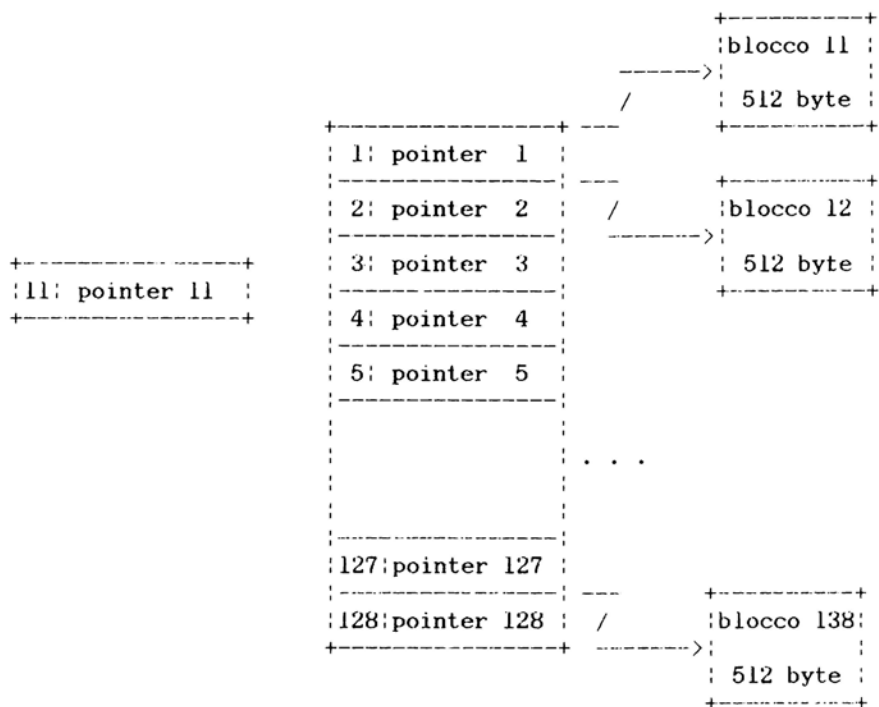
6- Il numero dei link al file, ovvero il numero di occorrenze, sotto qualsiasi nome, che il file ha nei vari directory.

7- L'indicazione se il file è a sua volta un directory, un file ordinario, un file speciale.

Per spiegare il meccanismo di allocazione dei file occorre prima illustrare la struttura logica delle unità di memorizzazione (dischi, nastri). Ogni volume fisico viene considerato composto di blocchi di 512 byte ciascuno accessibili singolarmente attraverso i loro indirizzi 0,1,... fino alla dimensione fisica del device.

Il file è quindi fisicamente allocato in blocchi non necessariamente contigui di 512 byte ciascuno. Lo I-NODE contiene gli indirizzi di questi blocchi. Per far fronte al problema di registrare la allocazione di file anche molto grandi, lo I-NODE dispone di un meccanismo di semplice, doppia o tripla indirezione, a seconda delle necessità. Più precisamente, lo I-NODE contiene per un file 13 indirizzi, i primi 10 dei quali puntano a blocchi fisici del file, l'undicesimo a blocchi di puntatori, il dodicesimo a blocchi di puntatori di puntatori, il tredicesimo a blocchi di puntatori di puntatori di puntatori.

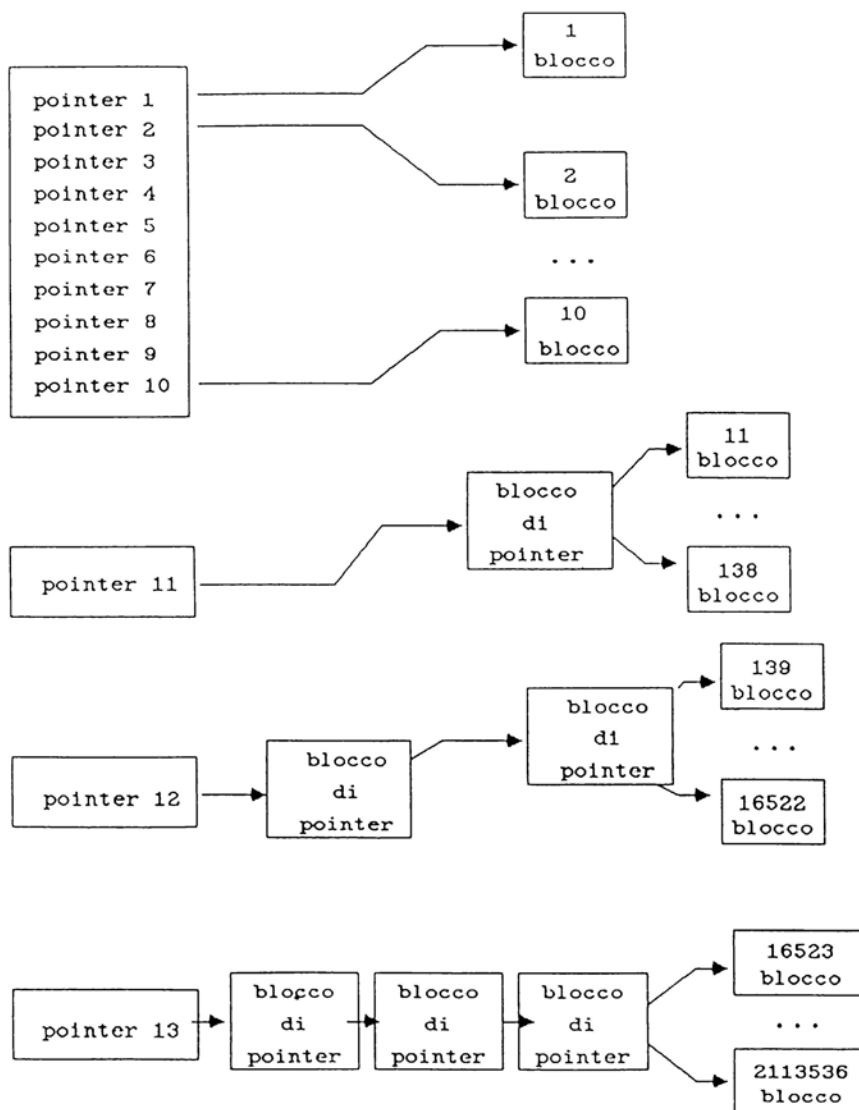
In altre parole, i primi 10 indirizzi puntano a blocchi di 512 byte ciascuno (non necessariamente fisicamente contigui):

```
                                            +----------+
                                            |blocco 1  |
                               ------------>|          |
+---------------+             /             | 512 byte |
| 1| pointer  1 | ------------              +----------+
|---------------|
| 2| pointer  2 | ------------
|---------------|            /              +----------+
| 3| pointer  3 |            -------------->|blocco 2  |
|---------------|                           |          |
| 4| pointer  4 |                           | 512 byte |
|---------------|                           +----------+
| 5| pointer  5 |
|---------------|
| 6| pointer  6 |
|---------------|
| 7| pointer  7 |
|---------------|
| 8| pointer  8 |
|---------------|
| 9| pointer  9 |
|---------------|
|10| pointer 10 | ------------
+---------------+            /              +----------+
                             -------------->|blocco 10 |
                                            |          |
                                            | 512 byte |
                                            +----------+
```

Se il contenuto del file si estende oltre la dimensione di 5120 byte, viene attivato un undicesimo indirizzo, che punta a sua volta ad un blocco di puntatori:

L'estensione raggiunta in questo modo è così di 5120 + (128 x 512) = 70656 byte.

Qualora il contenuto del file eccedesse questa dimensione, verrebbe attivato il 12 pointer, il quale estende lo stesso meccanismo al secondo anziché al primo livello, ovvero contiene blocchi di puntatori di puntatori. Ai 70.656 byte se ne possono aggiungere così altri

128**2 x 512 = 8.388.608

per complessivi 8.459.264 byte. Nei casi in cui il file eccede questo ulteriore limite, viene attivato il tredicesimo ed ultimo indirizzo, che si basa su un meccanismo di triplice indirizzamento, per cui contiene puntatori di un livello ulteriore

Struttura di un file

Estensione massima:
2.113.536 * 512 = 1.082.201.088 bytes

rispetto ai precedenti, per una aggiunta di

128**3 x 512 = 1.073.741.824 byte

i quali danno in totale una estensione massima di

1.082.201.088 byte

secondo la formula

[(10+128+128**2+128**3)x512]

Questo meccanismo comporta, in teoria, che si abbia un solo accesso al disco per i primi 5120 byte; due per quelli di numero compreso tra 5127 e 70.656; tre per quelli di numero compreso tra 70.657 e 8.459.264; quattro, infine, per quelli di numero maggiore di 8.459.264.

L'allocazione "sparsa" dei file comporta per gli utenti UNIX numerosi vantaggi. Non é necessario dichiarare al sistema la dimensione dei file che si vogliono creare, i file stessi possono crescere o diminuire di dimensioni a piacere. L'unico vincolo é rappresentato dalla capacitá fisica del device: non é ammesso che un file risieda in piú unita fisiche diverse. Le controindicazioni di questo sistema risiedono soprattutto nella velocitá: il trasferimento di soli 512 byte per volta e la distribuzione "random" dei blocchi sulle tracce del disco rallentano notevolmente l'accesso ai file.

UNIX sopperisce a questo limite strutturale costituendo in memoria centrale un pool di buffer (il cui numero é scelto in sede di configurazione del sistema, tipicamente fra 10 e 100). Quando un blocco é richiesto in lettura, lo si ricerca in primo luogo nell'ambito del pool. Quando la ricerca ha successo, i dati sono disponibili al programma utente senza che sia stato necessario eseguire alcuna operazione di I/O. Altrimenti il buffer meno recentemente usato viene scelto come area su cui trasferire i dati da disco. In modo simile ci si comporta per le operazioni di scrittura. Viene inoltre applicato un algoritmo di "read ahead" (lettura anticipata) che ha l'obiettivo di far trovare giá in memoria i dati che presumibilmente il programma richiederá (si pensi ad un programma che legge sequenzialmente un file).

La scrittura fisica dei buffer é anch'essa ritardata fino al

momento in cui il buffer é richiesto per ospitare dati diversi. Un processo di sistema (/etc/update) periodicamente provvede ad aggiornare i dati su disco. Il comando **sync** (ed una analoga system call) effettua la stessa operazione su richiesta dell'utente. E' evidente, tuttavia, che in caso di caduta del sistema (dovuta ad esempio a mancanza di corrente), il contenuto del disco sia normalmente non del tutto aggiornato, con possibile grave perdita di dati. L'aggiornamento dell'immagine dei file su disco é quindi in ultima analisi a carico dei programmi utente: queste funzioni si trovano normalmente nei programmi applicativi standard quali editor e data base management system.

3.2 LE CHIAMATE DI I/O.

Come abbiamo visto, le dimensioni di un file sotto UNIX non sono determinate altro che dal numero di byte effettivamente scritti in esso. Va inoltre notato che non si ha alcuna distinzione tra accesso random e sequenziale.

Quando un comando coinvolge un file, esso automaticamente lo "apre", a seconda dei casi, in lettura, scrittura o in entrambe. La chiamata di apertura di un file (che si suppone esista giá) puó essere schematizzata nella forma

FILE-DESCRIPTOR = open(nome-file, accesso)

Dei due argomenti della **open** il primo determina il file in questione, il secondo é un campo che specifica una apertura in lettura, scrittura o lettura/scrittura. La chiamata di **open** restituisce un valore intero che é detto FILE-DESCRIPTOR, che da quel momento identifica il file in questione per il comando che ha invocato la chiamata.

Analoga alla chiamata di **open** é quella di **create.** Con la stessa sintassi della precedente, essa crea il file in questione qualora non esista, o lo tronca a zero byte qualora esista giá. Come la **open** , anche la **create** fornisce un FILE-DESCRIPTOR di ritorno.

La scrittura e la lettura avvengono dunque, di massima, in

modo sequenziale. Il che vuol dire che se l'ultimo byte ad essere stato letto o scritto é l' n-simo, la successiva chiamata si riferirá all' n+1-esimo, ovvero al byte che lo segue immediatamente.

Le chiamate di lettura e scrittura possono essere schematizzate nel modo seguente:

```
n = read(FILE-DESCRIPTOR, BUFFER, NUMERO-DI-BYTE)
```

```
n = write(FILE-DESCRIPTOR, BUFFER, NUMERO-DI-BYTE)
```

Il numero **n** restituito dalle primitive di I/O contiene il numero di byte effettivamente trasferito: esso é quindi uguale a NUMERO-DI-BYTE in condizioni normali. Valori diversi segnalano condizioni di fine file o situazioni di errore.

Durante le operazioni di accesso a un file, é mantenuto attivo un puntatore, il quale indirizza al prossimo byte da leggere o scrivere. Quando si effettua un'operazione di lettura o scrittura che coinvolge n byte, il puntatore viene automaticamente incrementato di n.

Chiarito questo, resta evidente che per passare dalla lettura sequenziale a quella random é sufficiente gestire il valore del puntatore, in modo tale che esso indirizzi alla parte richiesta del file. Questa si realizza con la chiamata **lseek** , la quale rende una posizione:

```
puntatore = lseek(FILE-DESCRIPTOR, OFFSET, BASE).
```

Il puntatore associato al FILE-DESCRIPTOR viene incrementato di un numero di byte indicato da OFFSET. Questo puó essere fatto avendo come punto di riferimento l'inizio del file, la sua fine o la posizione attuale. Il punto di riferimento viene specificato da BASE. Si esaminino gli esempi che seguono:

lungh.del file	puntatore prima	OFFSET	BASE	puntatore dopo
3265	1350	100	0 (INIZIO)	100
3265	1350	100	1 (ATTUALE)	1450
3265	1350	-100	2 (FINE)	3165

Nel trattare le principali chiamate di I/O abbiamo sempre fatto riferimento al FILE-DESCRIPTOR, introdotto con le chiamate di **open** o **create** , anziché ad un nome di file. In effetti, la funzione delle chiamate **open** e **create** é proprio quella di trasformare il path-name del file in questione in un piccolo intero, il FILE-DESCRIPTOR appunto, al quale fare ricorso per gli ulteriori riferimenti allo stesso file. Quando viene aperto un file, viene aggiornata una tabella di sistema che contiene rispetto a quel file le seguenti informazioni:

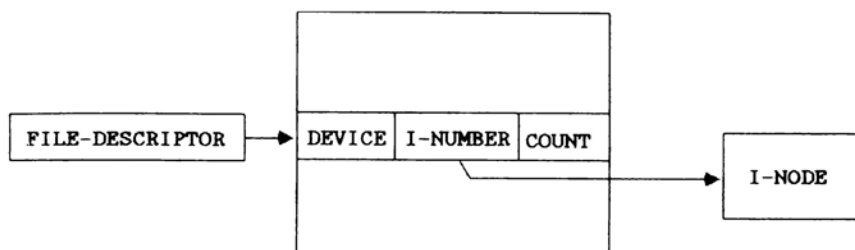
1- il suo device

2- il suo I-NUMBER

3- il suo puntatore di **read / write.**

Poiché, come si é visto in precedenza, lo I-NUMBER individua lo I-NODE del file, attraverso questa tabella di sistema sono rese disponibili tutte le informazioni relative al file in esame.

Il FILE-DESCRIPTOR, più volte citato in precedenza, non é altro che il puntatore alla tabella di sistema in questione.

Dunque, il FILE-DESCRIPTOR, attraverso il ricorso a tale tabella, é in grado di fornire in modo esaustivo tutte le informazioni relative al file su cui si sta operando.

3.3 LE PROTEZIONI

Un file viene associato a colui che lo ha creato, che quindi ne é a tutti gli effetti padrone. Quando un file viene creato, resta inoltre ad esso associata una successione di 10 bit, che ne specifica le protezioni. I tipi di protezione sono tre: da lettura, da scrittura, da esecuzione (ciò che ha senso, ovviamente, solo per determinati file). Le protezioni sono pensate per tre possibili utenti: il padrone del file (**owner**), un utente appartenente al suo stesso gruppo di lavoro, un estraneo. Poiché si hanno tre tipi di protezione per tre tipi di utente, combinatoriamente, si occupano 9 bit complessivamente:

PROTEZIONI PER L'OWNER DEL FILE			PROTEZIONI PER UN UTENTE DEL GRUPPO			PROTEZIONI PER UN ESTRANEO		
R	W	X	R	W	X	R	W	X
lett.	scrit.	esec.	lett.	scrit.	esec.	lett.	scrit.	esec.

R sta per **read** , W per **write** , X per **execute.** L' attributo X é significativo solo per i file eseguibili (programmi binari o shell) e per i directory. Quando un file é un directory, essi assumono il seguente significato:

R = abilita alla consultazione del directory

W = consente di creare e cancellare nuovi file nel directory

X = consente di consultare il directory, senza però accedere ai suoi file.

Con il comando **ls -l** si ottiene l'elenco dei file contenuti in un directory, con la specificazione delle relative protezioni.

L'esempio che stiamo analizzando mostra quali sono le protezioni standard, o di "default", sui file e sui directory:

1	2	3	4	5	6	7
-rw-r--r--	1	owner	2610	May 9 14:20	May 9 16:10	lettera
drwxr-xr-x	5	owner	423	May 10 10:20	May 11 16:20	documen
drwxr-xr-x	2	owner	344	May 9 10:20	May 9 10:20	ufficio

total 7 blocks

Il proprietario di un file o directory può ovviamente mutare a suo piacimento le protezioni in qualsiasi momento; tuttavia il sistema provvede, salvo esplicita istruzione in contrario, a rendere "leggibile" ogni file ad ogni utente del sistema.

Il diritto di "scrittura" nei file è invece normalmente riservato al proprietario; questo per evitare che qualcuno modifichi un file senza che ne sia avvertito, o che sia consenziente, il suo proprietario. Anche in questo caso, come nel precedente, è comunque possibile cambiare le protezioni standard, ossia abilitare altri a scrivere su uno o più dei propri file.

Il permesso di "eseguire" un file è sensato soltanto nel caso in cui quel file sia da usarsi come comando. Il permesso di "eseguire" un directory consente ad altri di entrare in esso con il comando **cd** , ma, come detto, non di accedere ai file.

Per modificare le protezioni definite per un file o directory, esiste il comando

chmod

seguito da "+" o "-" e poi r , **w** o **x** , a seconda delle protezioni che si vogliono attribuire, e dal nome del file o directory a cui debbono essere applicate.

Per esempio, se vogliamo che il nostro file "computer" sia disponibile in lettura a tutti gli utenti, comporremo

```
chmod +r computer
```

Se vogliamo che nessuno possa accedere al nostro directory nome-utente-2, comporremo

```
chmod -r {nome-utente-2}
```

Dopo un comando del genere, nessuno puó piú accedere a quel directory, o leggere i file immagazzinati in esso.

3.4 SIGNIFICATO DEL BIT SET-USER-ID

Un discorso a parte merita il primo bit dei 10 di protezione, che fino ad ora non abbiamo preso in considerazione. Il primo dei dieci bit, detto

SET-USER-ID

consente la protezione di un file in ambiente ordinario, e pure la sua disponibilitá rispetto a programmi privilegiati. Questo meccanismo consente un opportuno equilibrio tra i due poli estremi dell'accesso indiscriminato da una parte, e della impossibilitá di accesso dall'altra. La cosa diventa piú comprensibile sulla base di un esempio concreto. Si consideri per esempio il file che contiene la coda dei file di output in attesa di stampa inviati alla printer. E' chiaro che tale file deve essere protetto, perché altrimenti chiunque potrebbe alterare l'ordine dei job o cancellare le richieste di altri. Ma se tale file fosse completamente protetto, il programma di spool che inserisce in coda i riferimenti ai file da stampare, invocato da un utente qualunque, resterebbe nella impossibilitá di accedervi in scrittura, e non potrebbe dunque compiere la sua funzione.

Evidentemente non é sufficiente attribuire al programma di spool, che in UNIX corrisponde al comando **lpr** , il proprie-

tario **root**. Supponiamo infatti che lpr sia eseguibile anche da altri, secondo lo schema di protezioni seguente:

```
-rwxr-xr-x    root    lpr
-rw-r--r--    root    coda
```

Quando **lpr** entra in esecuzione, esso ha gli stessi diritti dell' utente che lo ha invocato, quindi non puó scrivere il file **coda**. Il meccanismo del SET-USER-ID risolve completamente questo problema. Al programma **lpr** che implementa la coda é associato dal proprietario (quindi in questo caso da root) un SET-USER-ID posto a 1. In questo caso quando **lpr** inizia l' esecuzione, l' utente viene temporaneamente modificato in quello del proprietario di lpr, e quindi root. Evidentemente i diritti di accesso del programma lpr diventano temporaneamente quelli di root, dunque il file **coda** diviene disponibile al programma invocato, nei tempi e nei modi della sua esecuzione, e rimane nel contempo protetto rispetto agli altri accessi diretti da parte dell'utente che ha invocato tale programma. Terminata l'esecuzione di lpr, l' utente ritorna quello iniziale, ed ogni diverso accesso alle risorse di root viene inibito.

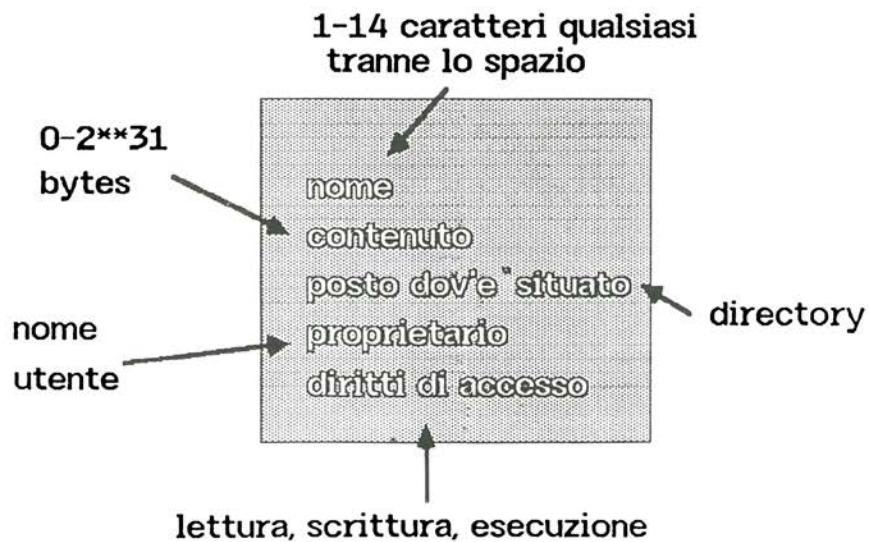
Il posizionamento del bit SET-USER-ID il cui significato possiamo ora interpretare nel senso "definisci come utente attivo il mio proprietario", é identificabile (nella lista prodotta da **ls**) da una "s" al posto della "x" che autorizza l' esecuzione da parte del proprietario del file.

```
-rwsr-xr-x    root    lpr
-rw-r--r--    root    coda
```

La maggior parte dei comandi di UNIX non ha necessitá di godere di particolari privilegi: fra quelli trattati in questo testo solo df, lpr, mail, mv, newgrp, ps, rmdir, su, mkdir hanno il SET-USER-ID posto a 1.

E' questo un altro tratto di UNIX che si richiama alla filosofia generale del "costruire sul lavoro degli altri": consentire l'uso di programmi di sistema da parte di qualsiasi utente, beninteso in modo razionale, in quanto non c' é sostanziale differenza tra i programmi di sistema e i programmi applicativi.

Nel sistema unix, ogni file ha:

CAPITOLO IV

IL LINGUAGGIO SHELL

4.1 LO SHELL

Lo SHELL presenta due aspetti fondamentali: per un verso esso é un linguaggio di comando, e dunque, prima di tutto, un'utile interfaccia tra l'utente e UNIX; per un altro, si é evoluto fino a costituire esso stesso, in tutto e per tutto, un linguaggio di programmazione di alto livello.

Ci occuperemo prima di tutto dello SHELL come interprete dei comandi. Nella sua applicazione più semplice, esso é dunque la chiamata di un comando. Tale comando puó essere un programma, ossia un file binario, oppure una procedura SHELL, concetto questo di cui ci occuperemo tra poco. Non fa alcuna differenza, sia per quanto riguarda l'utente che dal punto di vista del funzionamento interno, che il comando sia un programma di sistema o un programma scritto da un utente. Il comando invocato puó presupporre uno o più argomenti. Nella sua forma più semplice, dunque, una chiamata allo SHELL é una sequenza di parole separate da blank (ossia da uno spazio bianco o un tab), la prima delle quali é il nome del comando da eseguire, e le successive indicano gli argomenti di tale comando. Gli argomenti possono essere raggruppati in tre classi:

(1) le opzioni che non richiedono parametri;

(2) quelle che richiedono a loro volta uno o più argomenti;

(3) gli operandi, normalmente nomi di file.

Come si vedrá dagli esempi, le opzioni sono contraddistinte dal simbolo "-" premesso ad una o più lettere. Vediamo un esempio di ciascuno dei tre tipi citati. Rispetto a (1), possiamo fare riferimento al giá visto comando **ls** , che fornisce

l'elenco dei file del directory corrente. Con l'opzione -l , che non richiede argomenti, si ottiene un elenco dei file di ognuno dei quali sono specificate ulteriori informazioni, come le protezioni attive, il nome del proprietario, la lunghezza in byte, la data dell'ultima modifica e quella di creazione.

Consideriamo invece il comando

tar -c **-b16** {file} .

Il comando **tar** serve a scrivere e a leggere su o da un nastro. L'opzione -c significa "create", e consente di aprire il nastro in scrittura, cancellando l'eventuale contenuto precedente. L'opzione **-b** serve a specificare a quanti blocchi per volta deve avvenire la scansione dell'operazione. Ecco che **16** è un esempio del tipo (2) sopra visto, perché è un argomento che satura l'opzione **-b** e non direttamente il comando.

Infine, il caso (3), ovvero quello di un file su cui operare, è esemplificato da quasi tutti i comandi visti nei capitoli sul file system; ma possiamo rifarci ancora al semplice comando **ls**.
Se componiamo

ls -l {directory} ,

allora anziché i file del directory corrente, verranno elencati quelli dei directory specificato da {directory}.

La combinazione di queste possibilità dà luogo ad un gran numero di casi possibili. Dato che molti studiosi e molti utenti hanno contribuito allo sviluppo di UNIX, è difficile introdurre una regola fissa per l'ordine degli argomenti, il quale può presentare, quindi, qualche variazione da comando a comando. Un certo standard tuttavia esiste, ed è così riassunto dalla Bell:

1 - I nomi dei comandi devono essere di lunghezza compresa tra i 2 e i 9 caratteri;

2 - I nomi dei comandi devono contenere solo lettere minuscole e cifre;

3 - I nomi delle opzioni devono essere di un solo carattere; separati tra di loro dalla virgola, oppure devono essere racchiusi tra " e separati da spazi;

4 - Tutti i nomi di ozioni devono essere preceduti da "-";

5 - E' possibile raggruppare più opzioni dopo il carattere "-";

6 - L'argomento relativo ad una specifica opzione deve essere separato dal nome di quella da uno o più spazi;

7 - Se un'opzione prevede un argomento, la presenza di quest'ultimo deve essere obbligatoria;

8 - Se un'opzione prevede più argomenti, questi devono essere

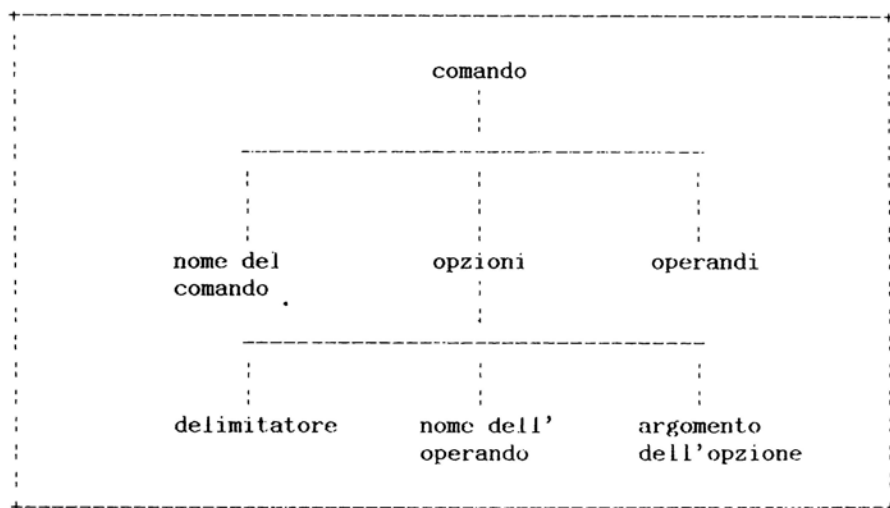
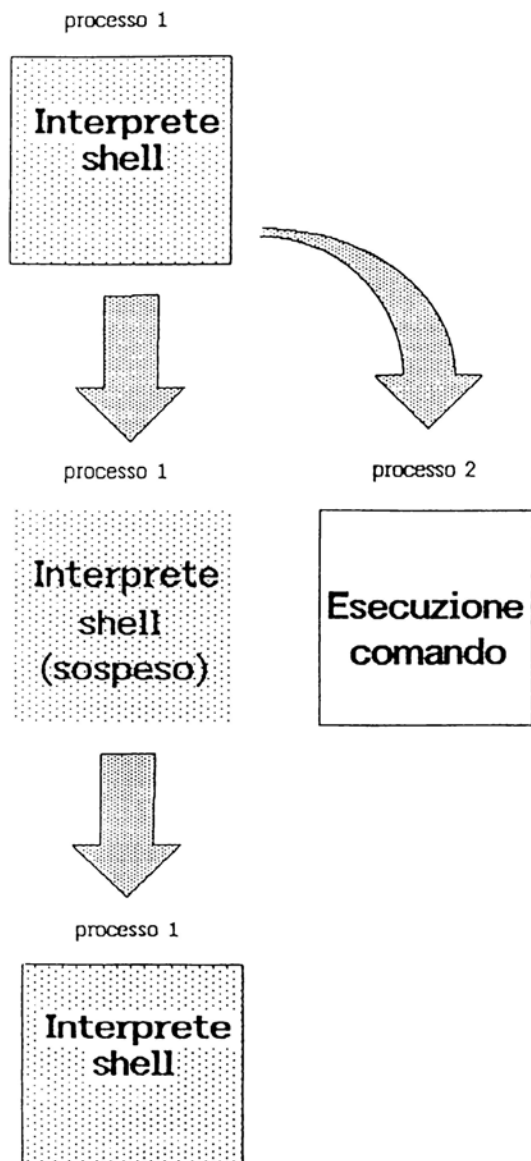
9 - Le opzioni devono precedere gli operandi;

10 - La stringa "--" può essere usata per indicare la fine delle opzioni;

11 - L'ordine delle opzioni non deve avere influenza sull'esecuzione del comando;

12 - L'ordine degli operandi può essere significativo, e tale significatività dipende dallo specifico comando;

13 - Il carattere "-" da solo sta a significare lo standard input.

Possiamo riassumere la situazione con il seguente schema:

ESECUZIONE DI UN COMANDO SHELL

Lo SHELL consente l'uso di particolari metacaratteri per rendere più sintetica la forma di un comando. Supponiamo di dovere passare come argomenti i nomi:

file1 file2 file3

Possiamo riferirci a tutti e tre contemporaneamente grazie alla stringa

file?

Infatti il carattere speciale

?

sta per un qualsiasi singolo carattere.

Il metacarattere

*

sta invece per qualsiasi singolo carattere o stringa. Così

mv * /u/maurizio/lettere

muoverá tutti i file del directory corrente nel directory /u/maurizio/lettere. Anche il metacarattere * puó essere collocato all'interno di una stringa. Se nel directory attuale esistono i file

lett.01 lett.12 lettere

con la stringa

let*

mi riferiró a tutti e tre i file contemporaneamente.

Una ulteriore possibilitá é offerta dal racchiudere i caratteri a cui ci si vuole riferire tra parentesi quadre:

[ab]

consentirá di riferirsi tanto ad **a** quanto a **b** Invece

[a-z0-9]

si riferirá a qualsiasi carattere compreso tra **a** e **z** o tra **0** e **9.** Poiché questo é un modo di riferirsi a più caratteri, ma presi uno alla volta, deve essere usato all'interno di una stringa.

Più avanti ci occuperemo in dettaglio della successione dei vari passi che si hanno quando si esegue una chiamata allo SHELL; ma notiamo fin da ora che, prima dell'esecuzione del comando, i metacaratteri sopra visti vengono espansi, ossia ad essi vengono sostituiti tutti i possibili valori, i quali vengono poi passati come argomenti al comando uno alla volta. Nel caso in cui non esistano elementi che soddisfano alla metanotazione, allora la stringa contenente i metacaratteri viene passata essa stessa come argomento.

Riepiloghiamo quanto detto fornendo una tabella di quello che si chiama in gergo **pattern matching** , vale a dire di soddisfacimento di un oggetto ai requisiti di un certo modello; nel caso specifico, essa servirá per stabilire quando una determinata stringa viene ad essere compresa nell'espansione di una metanotazione.

REGOLE DI MATCHING	
*	denota ogni stringa, compresa quella nulla
?	denota ogni singolo carattere
[c1c2...cn]	denota ogni singolo carattere c1 ... cn, preso singolarmente.
[a-z]	denota ogni carattere compreso tra a e z
\c	denota precisamente il carattere che segue il controslash, preso alla lettera anche se e' un metacarattere

4.2 REDIRECTION DELL'I/O

L'input e l'output standard di un comando sono di solito il terminale stesso attraverso il quale il comando è stato impostato. Così, impostando il comando

who

l'elenco degli attuali utenti del sistema viene proposto nel video del terminale dalla cui tastiera è stato inviato il comando. Esistono tuttavia alcuni segni convenzionali che lo SHELL riconosce come opzioni per sè, e che non trasmette al comando come argomenti. Uno di questi provoca l'effetto di sostituire al normale file di input o di output un file deciso dall'utente. Così impostando

who > lettere

lo SHELL riconosce il carattere speciale " > ", e non lo passa come argomento al comando **who.** Invece, esso provvede ad indirizzare l'output del comando **who** sul file "lettere", che viene sovrascritto qualora già esista e viene creato qualora non esista. Analogamente può essere sostituito lo standard input di un comando. Per esempio, il comando

wc

conta il numero di caratteri, di parole e di righe di ciò che ha in input. Ora, se si imposta

wc < lettere

lo SHELL provvede a far prelevare dal comando i dati, in lettura, dal file "lettere", e a fornire il suo output, poichè nessuna redirezione è stata specificata, sul terminale da cui è stato chiamato.

A proposito della redirezione dell'output, va ricordato che, cosí come é possibile, nei modi visti precedentemente, ottenere l'uscita sovrascrivendo il contenuto di un file esistente, allo stesso modo é possibile anche aggiungere in fondo al contenuto di un file. Per ottenere questo risultato, basta semplicemente raddoppiare il segno speciale di redirezione dell'output. Cosí

ls >> lettere

aggiungerá in coda al file "lettere" l'uscita del comando **ls** , senza intervenire sul contenuto precedente del file. E' da tenere presente che i segni speciali " > " , " >> ", e " < " sono gestiti in generale dallo SHELL, e non dal particolare comando eseguito. Ad esempio, il comando

cat

seguito dal nome di un file, porge tale file a video; ma con

cat file1 file2 file3 > file4

al video si sostituirá file4, ovvero il comando concatenerá i file 1, 2 e 3 in file4, sovrascrivendo il contenuto precedente di quest'ultimo. Viceversa

cat file1 file2 file3 >> file4

appenderá in fondo a file4, ordinatamente, file 1, 2 e 3.

Abbiamo visto che lo standard input, salvo esplicita redirezione, é associato alla tastiera. Se dunque diamo un comando che presuppone la presenza di un operando, e non forniamo un nome per tale operando, esso viene automaticamente identificato dal sistema come lo standard input. Dunque ció che di seguito comporremo a tastiera verrá interpretato come il file di input. Il comando opererá su di esso quando troverá un marc di fine file, ossia un **EOF** , normalmente [CTRL] d.

In merito alle redirezioni dell'input e dell'output, vediamo ora la modalitá di uso di quello che é detto l' **here input.**

```
cat > file4 <<!
cantami o
diva del
pelide achille
!
```

La costruzione sintattica precedente (che normalmente è utilizzata all' interno di una procedura SHELL) indica di considerare come input le righe che seguono il comando, fino all' occorrenza del carattere "!" che agisce come EOF.

```
+----------------------------------------------------------------+
|                                                                |
|           Si provi a dare il  comando  cat  senza alcun        |
|           argomento.                                           |
|                                                                |
|           Il cursore si posizionera' a capo, senza dare        |
|           il segno di prompt.                                  |
|                                                                |
|           Si compongano ora alcune parole.                     |
|                                                                |
|           Si dia un [CTRL] d.                                  |
|                                                                |
|           A questo punto verra' presentato l'output di cat,    |
|           cioe' ricompariranno a video le parole composte in   |
|           precedenza.                                          |
|                                                                |
+----------------------------------------------------------------+
```

4.3 FILTRI E PIPE

Come si è più volte detto, uno dei concetti portanti di UNIX è quello della concatenabilità dei comandi. Un programma che fornisce il suo output come input a un altro programma è detto **filtro**. In generale, ogni comando di UNIX può essere usato come filtro.

Perché sia possibile la comunicazione tra due programmi, sotto qualsiasi sistema operativo, si deve fare riferimento, in linea di massima, a un file di appoggio. Così un primo programma, che possiamo per comodità chiamare **A** , scriverà un certo file, e un secondo programma, che chiamiamo **B** , leggerà quello stesso file. Un tratto caratteristico di UNIX è quello di prevedere in modo sistematico questo collegamento, scavalcando la necessità del file intermedio. Sotto UNIX, più precisamente, abbiamo visto che un file di quel tipo viene detto **pipe**. Un **pipe** non è altro che un canale di collegamento privilegiato tra due processi, realizzato attraverso un buffer allocato in memoria centrale, e concepito in modo tale che il primo dei due programmi si arresta quando esso è saturo, e il secondo si pone in stato di attesa quando esso è vuoto. Il pipe , una volta esaurita la sua funzione, cessa di esistere, e non impone quindi all'utente l'onere di cancellare file di appoggio temporanei.

Il segno speciale che in ambiente SHELL consente di collegare due comandi attraverso un **pipe** è la barra verticale:

¦

Come esempi, possiamo considerare i seguenti. Abbiamo a disposizione un comando che ordina un file di stringhe di input. Tale comando è **sort.** Se allora impostiamo

who ¦ sort

otterremo che il comando **who** , che viene eseguito per primo, consegni il suo output, ovvero l'elenco degli attuali utenti del sistema, al comando **sort** attraverso un **pipe.** Il secondo

comando, a questo punto, esegue la sua funzione restituendo allo standard output, ovvero al terminale chiamante, l'elenco ordinato degli utenti.

Naturalmente il processo è iterabile. Così è possibile impostare l'espressione, ad esempio,

ls ¦ grep UNIX ¦ wc -l

Il comando **ls** produce l'elenco dei file sotto il directory corrente. Tale output è passato come input al comando **grep** , il quale ha come argomento la stringa "UNIX". Il comando **grep** seleziona nell'ambito del suo input tutte le righe in cui occorre una certa stringa che gli è data come argomento, nel caso specifico la parola "UNIX". Il **grep** agisce a sua volta come un filtro, e passa il suo output, che sará quindi una successione di righe, attraverso un nuovo **pipe** , al comando **wc** , che ha come opzione -l , ovvero è richiesto di contare solo le righe, e non i caratteri e le parole. L'esito di un comando composto come sopra sará in definitiva quello di fornire il numero di file elencati sotto il directory corrente che contengono la stringa "UNIX".

Un altro comando di una certa utilitá in connessione allo SHELL è

echo

Esso non fa altro che presentare in output i suoi argomenti, separati da blank. Questo è dunque il modo per far sì che lo SHELL possa presentare messaggi di display. Per esempio, l'esecuzione del comando

echo UNIX

produrrá come output il display a terminale di

UNIX

Un'altra opzione importante dello SHELL é realizzata con il carattere speciale

&

Se la chiamata ad un comando é seguita da tale carattere, l'esecuzione avviene in background, ovvero senza che lo SHELL attenda che il programma abbia terminato di girare per rendere il controllo all'utente.

Va infine notato che nelle operazioni iniziali, prima ancora di dare il controllo alle richieste di comandi da terminale, lo SHELL cerca nel login directory se esiste un file di nome ".profile"; in caso affermativo, esso viene interpretato come un elenco di comandi, i quali vengono eseguiti. Questo rende possibile predisporre in modo facile un comando iniziale, da eseguirsi ad apertura di sessione, come ad esempio il display di un menu.

ESECUZIONE IN BACKGROUND

comando argomento ... argomento &

processo 1

Interprete shell

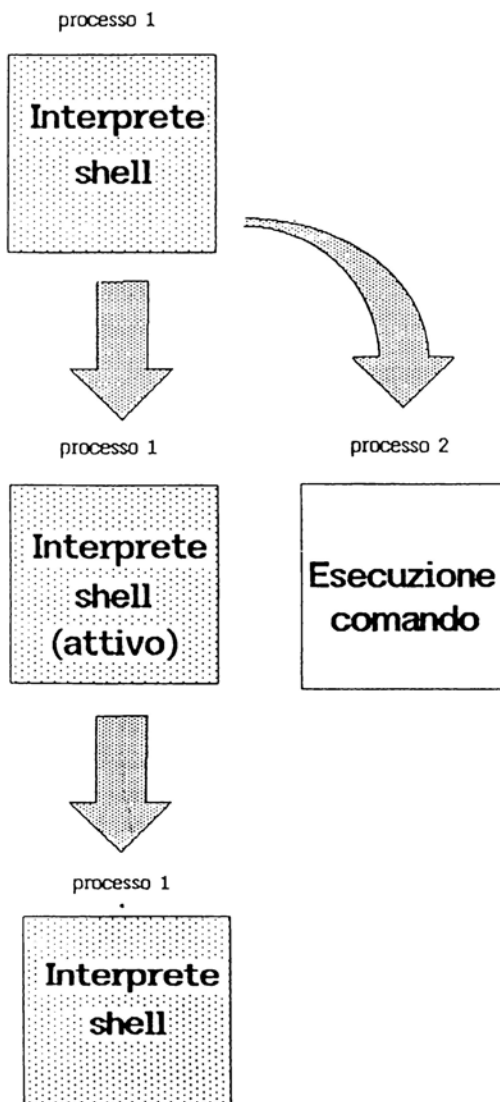
processo 1

Interprete shell (attivo)

processo 2

Esecuzione comando

processo 1

Interprete shell

4.4 LE PROCEDURE SHELL

Puó essere di estrema utilitá immagazzinare in un file i comandi che si vuole lo SHELL esegua in successione. Un tale file si dice una PROCEDURA SHELL. La catena di comandi che esso contiene viene mandata in esecuzione con la chiamata

sh {nome-file}

Poiché un comando puó in generale sopportare degli argomenti, é necessario potere "passare" allo SHELL tali parametri. Essi vengono rappresentati, all'interno di una PROCEDURA SHELL, dai simboli

$1, $2, $3, ...

In questo formato vengono assunti come posizionali, ovvero il numero **n** che segue il segno $ indica che quello é l' **n-simo** parametro nell'ordine di entrata.

A proposito dei parametri di SHELL va ancora notato che:

$0 e' il nome del file in esecuzione -

$# e' il numero dei parametri posizionali della procedura -

$* si riferisce a tutti i parametri posizionali eccetto $0 -

Supponiamo di avere creato un file il cui contenuto sia semplicemente

grep $1 $2

e il cui nome sia "cerca". Se ora impostiamo

sh cerca paolo file1

lo SHELL, chiamato da **sh** , riconoscerá il nome "cerca" come quello di un file, leggerá il comando ivi contenuto, e gli fornirá come argomento, al posto del parametro $1, la stringa "paolo". Il risultato sará costituito da tutte le linee di file1 le quali contengano la stringa "paolo".

Naturalmente queste facility sono combinabili con tutte quelle che abbiamo viste in precedenza. Possiamo quindi legare una PROCEDURA SHELL con un ulteriore comando attraverso un **pipe** , e impostare ad esempio

sh cerca paolo | sort

oppure

sh cerca paolo >> lettere

Con l'istruzione

chmod a+x cerca

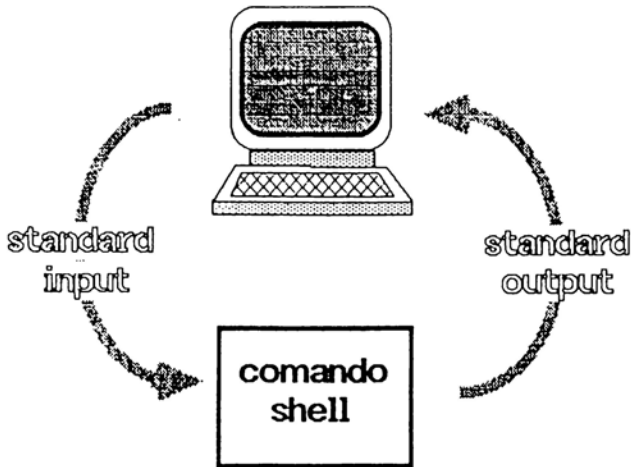
si definisce "cerca" come file eseguibile per il sistema. Da questo punto in poi si puó invocare il comando, quindi, sia componendo semplicemente il suo nome, come nel caso di

cerca paolo file1

sia facendo una chiamata allo SHELL:

sh cerca paolo file1

Questo fatto determina il vantaggio che le procedure SHELL e i programmi possono essere usati in modo del tutto indifferente, e si configurano quindi come pienamente interscambiabili.

RIDIREZIONE DELL'INPUT / OUTPUT

4.5 LE VARIABILI DELLO SHELL

Lo SHELL consente l'uso di variabili tipo stringa. Una variabile puó essere un nome qualsiasi, purché inizi con una lettera, e sia composta di lettere, cifre e sottolineature. L'assegnazione di un valore a una variabile si realizza con il carattere "=":

```
user=paolo
```

Per ottenere il valore di una variabile, é sufficiente riferirsi al suo nome premettendogli il segno $. Cosí

```
echo $user
```

risponderá

```
paolo
```

Poiché gli spazi sono significativi nelle espressioni, puó capitare di costruire espressioni ambigue. Una comoda opzione che risolve diversi casi di ambiguitá é la possibilitá di racchiudere il nome di una variabile tra parentesi graffe, quando al suo valore va aggiunta una lettera o una cifra. Cosí, se

```
paolo=/u/paolo/cre ,
```

$paolo e ${paolo} avranno lo stesso valore, ma

```
$paoloa
```

sará il valore della variabile "paoloa", mentre

```
${paolo}a
```

sará uguale a

/u/paolo/crea .

Naturalmente noi possiamo mettere in una variabile anche un comando con i suoi argomenti. Cosí potremo fare l'assegnazione:

a="cat filel file2 file3"

Se ora diamo il comando:

$a > file4

otterremo l'esecuzione di:

cat filel file2 file3 > file4

L'insieme delle variabili SHELL viene chiamato **environment** del processo. Con il comando **export** si dichiarano le variabili che si vuole siano disponibili agli eventuali sottoprocessi chiamati. Cosí

export paolo a

consente di trasmettere, ogni volta che un sottoprocesso viene attivato, i valori delle variabili **paolo** e **a.**

Le variabili di sistema nella tabella seguente sono tutte dichiarate esportabili dal sistema stesso.

Un altro potenziamento dell'ambiente SHELL é dato dal fatto che un comando racchiuso tra i segni

‘ ‘

viene eseguito immediatamente, e il risultato dell'esecuzione puó venir passato come argomento a un altro comando. Supponiamo ad esempio di mettere in un file un elenco di nomi di file che vogliamo cancellare. Chiamiamo tale file con il nome di **listafile.** Diamo poi il comando

rm ‘cat listafile‘

Ció che avviene é che per prima cosa viene eseguito il comando cat listafile. L'output viene poi passato come argomento al comando **rm** , il quale riceve quindi come argomenti i nomi dei file da cancellare.

VARIABILI SHELL (1)

$# Contiene il numero degli argomenti trasmessi alla procedura in corso.

$* Tutti gli argomenti dello SHELL.

$- Opzioni invocate per lo SHELL.

$? Fornisce il codice di ritorno dopo la chiamata ad un comando. Il codice di ritorno é zero quando l'esecuzione é stata priva di errori, mentre l'interpretazione dei valori diversi da zero varia a seconda del comando. Di solito essa é uguale a 200 piú lo stato. Si veda in un manuale il comando **signal** per avere un elenco dei valori di stato.

$$ Process-id del processo corrente.

$! Process-id dell'ultimo processo mandato in background.

$HOME Reca il nome del login directory dell'utente. E' a tale variabile che fa riferimento il comando **cd** quando riceve come argomento un nome che non cominci con /. Questa variabile é presente ovviamente nel profile di login dell'utente.

$PATH E' un elenco di nomi di directory, separati da :, nei quali lo SHELL cerca i comandi da eseguire. Questa variabile viene predisposta, di solito, in modo che lo SHELL cerchi

1) nel directory corrente;

2) in /bin;

3) in /usr/bin.

Ogni utente puó comunque stabilire in quali directory, e in che ordine, lo SHELL deve cercare i comandi che riceve, predisponendo la variabile $PATH che conviene al suo caso. Tale istruzione, se la sua validitá si estende in genere senza limitazioni, puó essere posta nel .profile dell'utente.

VARIABILI SHELL (2)

$IFS Contiene i caratteri assunti come separatori di parole negli argomenti (normalmente **blank, tab, newline**).

$MAIL Contiene il nome di quel file che, se é stato modificato, provoca il messaggio "you have mail"; Serve quindi per gestire la posta elettronica.

$PS1 Segno di **prompt** , per default **$**

$PS2 Segno di **prompt** secondario, per default >

4.6 GLI STEP DI ESECUZIONE DI UN COMANDO

Ora che abbiamo visto una panoramica abbastanza vasta delle possibilitá offerte dall'ambiente SHELL, vediamo di riepilogarla analizzando i vari passi in cui si articola l'esecuzione di un comando.

1 - Per prima cosa vengono eseguiti i comandi racchiusi tra ``.

2 - Avviene poi la sostituzione delle variabili con il loro valore.

3 - Si ha l'individuazione degli argomenti: essi vengono riconosciuti scandendo la riga di comando, e riscontrando i separatori (che sono contenuti nella variabile $IFS).

4 - Ricerca dei nomi di file. E' da notare che, se un pattern non viene riconosciuto, allora esso rimane inalterato, ovvero il nome é passato come tale.

5 - Apertura dei file e allocazione in accordo con le redirezioni richieste.

6 - Trasferimento delle variabili di environment esportabili al programma chiamato.

7 - Ricerca del comando nei directory secondo la variabile $PATH.

8 - Creazione di un nuovo processo. Questo provoca lo stato di attesa del terminale da cui il processo é chiamato, salvo che esso non sia stato chiamato in background (&). In questo secondo caso, continuazione del colloquio con il terminale.

E' da notare che, qualora si debba fare uso caratteri speciali, che non si vuole siano interpretati, é possibile citarli premettendo il segno \ , nel qual caso si cita il singolo carattere, oppure, se si ha a che fare con molti caratteri, racchiudendoli tra i segni ''. Va notato infine che la citazione fra doppie virgolette (" "), é anch'essa effettuabile, ma non é efficace nei passi 1 e 2, ma solo nei passi 3 e 4 spiegati sopra.

4.7 LO SHELL COME LINGUAGGIO

Come abbiamo visto, lo SHELL gestisce l'uso di variabili, sia stringa che numeriche. Per di piú, nello SHELL é possibile una completa gestione del controllo, alla stregua di ció che avviene in ogni linguaggio di programmazione evoluto. Per gestione del controllo intendiamo qui la possibilitá di realizzare salti condizionati e strutture di ripetizione.

Analizziamo ora queste caratteristiche dello SHELL, specificandone la sintassi.

Per verificare che lo SHELL possiede tutta l'ampiezza di un linguaggio di alto livello, basterá fare vedere come esso possa esprimere le tre strutture fondamentali della sequenza, della alternativa e della ripetizione, poiché é ben noto che qualsiasi algoritmo é esprimibile facendo ricorso a quelle sole (cosí detto teorema di Boehm-Jacopini).

Mostrare che la sequenza é realizzabile sotto SHELL é banale. Infatti, come abbiamo visto, una procedura SHELL esegue sequenzialmente i comandi che trova elencati in un file.
Occupiamoci pertanto dell'alternativa. Va detto prima di tutto

che lo SHELL si richiama frequentemente, nella sua sintassi, al comando

test

Il comando test non fa parte propriamente del linguaggio SHELL, ma è da questo richiamato e sfruttato frequentemente nelle strutture condizionali. Possiamo dire che il comando test ha come argomento un certo predicato, e restituisce un valore booleano in dipendenza del verificarsi o meno dello stesso. Ad esempio:

test stringa	e' vero quando la variabile "stringa" non e' space.
test -f filel	e' vero quando filel esiste
test -d filel	e' vero quando filel e' un directory
test -r filel	e' vero quando filel non e' protetto da lettura
test -w filel	e' vero quando filel non e' protetto da scrittura .

La sintassi dell'alternativa è la seguente:

```
if   comandol
then comando2
else comando3
fi
```

Lo SHELL esegue comandol, controlla il suo codice di ritorno, e in dipendenza da quello esegue comando2 o comando3.

Un esempio tipico può essere il seguente:

```
if  test -f lettere
then cat paolo >> lettere
else cp paolo lettere
fi
```

In questo esempio lo SHELL controlla l'esistenza di un file di nome "lettere" raggiungibile dal directory attuale; se esso esiste vi appende in fondo il file "paolo", mentre se non esiste lo crea e vi copia il contenuto di "paolo".

Quando vi sono molti if nidificati, anziché scrivere nel modo seguente

```
if comando1
then comando2
else if comando3
     then comando4
     fi
fi
```

la sintassi dello SHELL consente la forma abbreviata:

```
if comando1
then comando2
elif comando3
then comando4
fi
```

Un modo tipico di realizzare una serie di alternative sotto SHELL è quello di sfruttare la forma **case**.
Eccone la sintassi:

```
case $i in
   modello1) comando1 ;;
   modello2) comando2 ;;
esac
```

Lo SHELL confronta la stringa che riceve in entrata con i vari modelli in successione; se trova coincidenza tra l'input e un modello, esegue il comando connesso con quest'ultimo.

Ad esempio

```
case $# in
   1) cat >> $1 ;;
   2) cat >> $2 < $1 ;;
   *) echo "questo comando presuppone 1 o 2 parametri" ;;
esac
```

Si ricorderá che la variabile $# assume come valore il numero dei parametri che la procedura SHELL riceve in input. Ora, questa procedura ammette solo due possibilitá: che si dia un solo parametro, e allora concatena lo standard input in coda al file specificato da tale parametro; che si diano due parametri, e allora concatena il primo file in coda al secondo. Se la procedura non riceve argomenti, o ne riceve piú di due, allora essa non esegue nessuno dei due comandi di concatenazione, ma stampa un messaggio di errore. L'asterisco come al solito significa "qualsiasi cosa". Ma se la procedura riconosce di essere in uno dei due casi precedenti, non esegue questo comando.

Nell'ambito del **case** é possibile porre due modelli con cui lo SHELL deve fare il confronto rispetto ai parametri di input in disgiunzione logica tra di loro, in modo tale che il comando connesso venga eseguito se il confronto é positivo per almeno uno dei due. Se si vuole che un certo comando venga eseguito quando il parametro di entrata é uguale ad **A** oppure a **B** , si deve porre una barra verticale, conforme al segno di **pipe** , ma con tutt'altro significato, tra i due modelli ammessi:

```
case $1 in
    A | B) comando1 ;;
    C) comando2 ;;
esac
```

A proposito dell'uso della citazione, che puó essere utile soprattutto in relazione ai modelli di riferimento del **case** , va ricordato che sotto UNIX essa puó essere realizzata in tre modi: con le semplici e con le doppie virgolette, e con il

carattere speciale

Quest'ultimo si riferisce esclusivamente al singolo carattere che lo segue immediatamente.

Per quanto riguarda la ripetizione, o loop, vi sono due modi fondamentali di realizzarla. Come per l'alternativa abbiamo visto il caso dell' "if ... then ", che controlla il codice di ritorno, e il **case** , che é gestito da una variabile ricevuta dall'esterno, cosí qui si hanno due loop analoghi, il primo fondato sul test del codice di ritorno di un comando ("while ... do"), il secondo gestito dal valore di una variabile ("for ... do").

La sintassi della prima forma di ripetizione é la seguente:

```
while comando1
do comando2
done
```

Il **while** testa il codice di ritorno di comando1, e fino a che esso é zero (sinonimo di successo) esegue comando2. Quando il codice di ritorno di comando1 é diverso da zero il loop termina.

La sintassi del secondo é

```
for i in x1 x2 x3 ....
do comando $i ; done
```

o, equivalentemente

```
for i in x1 x2 x3 ...
do
comando1
comando2
...
done
```

Il punto e virgola é richiesto solo quando si continua a scrivere sulla stessa riga.

I comandi racchiusi fra il **do** e il **done** vengono eseguiti per ogni valore x1 x2 x3 ...., associato alla variabile $i. Esiste anche una forma sintetica del comando

```
for i do comando $i ; done
```

nella quale la variabile $i assume i valori dei parametri di ingresso.

Si puó forzare l'uscita da un loop, for o while che sia, attraverso l'istruzione

```
break.
```

L'istruzione

```
continue
```

provoca invece attiva la successiva iterazione.

Ecco quindi che, come si é visto, ognuna delle tre strutture fondamentali é realizzabile sotto SHELL.

Uno dei punti di forza dello SHELL é che esse sono combinabili a piacere. Vediamo un esempio di una ripetizione che racchiude al suo interno una serie di alternative espresse con il **case** :

```
for i
  do case $i in
    1) comando1 ;;
    2) comando2 ;;
    *) echo 'valore fuori limite' ;;
  esac
done
```

Da una procedura SHELL é sempre possibile attivarne un'altra semplicemente nominandola, come si fa in generale con i comandi di UNIX. Nell'esempio precedente, comando1 potrebbe essere altrettanto bene un comando UNIX, un programma utente eseguibile, o una procedura SHELL. L'esecuzione della procedura chiamante riprenderá, dopo l'esecuzione della procedura chiamata, dall'istruzione successiva.

La terminazione di una procedura avviene quando lo SHELL incontra la fine del file dei comandi, oppure su esplicita richiesta, attraverso l'istruzione

exit

4.8 IL DEBUGGING

Il debugging delle procedure SHELL si fonda su tre opzioni fondamentali:

1 - set -n

2 - set -v

3 - set -x

La 1 serve per provocare l'interruzione dell'esecuzione ad un punto voluto. I successivi comandi saranno quindi solo letti dallo SHELL, fino al primo EOF che esso incontra. Se ad esempio impostiamo dal nostro terminale

set -n

il terminale rimarrá inattivo fino a che non impostiamo un e.o.f.

La 2 permette il display successivo delle istruzioni, a mano a mano che lo SHELL le legge.

La 3, infine, fornisce la **trace** : vengono presentati i comandi via via eseguiti, e posti in evidenza i valori dei loro argomenti.

Per annullare le opzioni, basta impostare

set -

Il valore delle opzioni attive é disponibile come $-.

4.9 ESERCIZI RIEPILOGATIVI SULLO SHELL

Analizzare le seguenti procedure, e rendersi conto di tutte le istruzioni ivi contenute. Cosa fanno queste procedure SHELL?

```
for i in man1 man2 man3 man4 man5 man7 man8
do      cd /u/man.III/$i
        if test -f $1.*
           then
                comando=`ls $1.*`
                stty page length 20 2>/dev/null
                nroff -man $comando
                stty -page length 0 2>/dev/null
                exit
            fi
done
echo man3: non ho trovato il comando $1
```

```
if test "$1" = "-h"
then echo '
Il comando "skulker" permette di cancellare tutti i file
"*.bak", "a.out", "core", "proof" e "galley" presenti nel proprio
directory o in un sub-directory.

Dopo avere dato il comando "skulker" il programma esplora tutti i
directory e quando trova uno dei file suddetti emette il
messaggio:

        < rm ... /u/utente/nome_file >?

Se si risponde con "y" il file sara' rimosso altrimenti no.
'
else find $HOME ( -name a.out -o -name core -o
        -name proof -o -name .*.bak
        -o -name galley  -o -name '*.bak' ) -ok rm {} ;
fi
```

Vogliamo ora sperimentare praticamente un caso di redirezione dell'I/O. Potremo sfruttare la procedura **skulker** ora vista.

Vogliamo farla girare in modo tale che essa prelevi il suo input da un file anziché da terminale. Dunque:

1 - Contare quanti file verranno cancellati dalla procedura -

2 - Creare un file con le risposte affermative alle domande che la procedura fará -

3 - Istruire lo SHELL a prelevare il suo input dal file cosí creato -

4 - Far girare la procedura -

5 - Verificarne l'esito listando i file rimasti.

Vogliamo realizzare un **pipe.** Vogliamo anche operare la redirezione dell'output dell'ultimo comando eseguito su un file che esista giá, in modo tale che l'output venga appeso in coda a tale file.

1- Creare un file e scrivere in esso qualche cosa -

2- Scegliere due comandi a piacere per connetterli con un **pipe** -

3- Istruire lo SHELL in modo che l'output dei **due** comandi connessi con il **pipe** sia appeso in coda al file creato -

4- Verificare che il precedente contenuto del file non sia stato alterato, e che l'output prodotto sia quello richiesto.

Una delle applicazioni piú tipiche dello SHELL é quella di realizzare dei **menu.** Si scelgano alcuni comandi, e si predisponga un menu, il quale fornisca gli opportuni messaggi, e mandi in esecuzione il comando scelto dall'utente.

CAPITOLO V

UNIX E OFFICE AUTOMATION

5.1 UNIX E OFFICE AUTOMATION

Come si é fatto rilevare nella introduzione generale, il sistema operativo UNIX é particolarmente adatto a risolvere due tipi di problemantiche: quella della produzione del software, nell'ambito della quale il suo cavallo di battaglia é la concatenabilitá dei programmi, e quella dell'automazione del lavoro di ufficio. Il presente capitolo é dedicato agli aspetti principali di questa seconda applicazione, e ai suoi strumenti essenziali, come la

posta elettronica e gli **editor.**

Nell'ambito del lavoro di ufficio, come é stato notato recentemente, si ha sempre meno bisogno di spostamenti fisici di ben precisi oggetti materiali. Ció che viene in linea di massima scambiato, e che costituisce, sotto varie forme, la materia prima del lavoro di ufficio, é, in definitiva, **linguaggio.** Se a tutta prima tale affermazione puó sembrare una esagerazione, tale essa non sembra piú di certo ad una analisi approfondita, sol che si ponga mente al tentativo di addurre controesempi. Linguaggio sono le informazioni che ci si scambiano, linguaggio le telefonate, le relazioni, le conferenze, i documenti, le lettere, gli atti ufficiali, i seminari d'aggionamento... Uno spostamento fisico di oggetti che non si risolvano immediatamente in linguaggio viene fatto normalmente in quanto tali oggetti sono strumenti indispensabili alla creazione di linguaggio: le penne, i timbri, la carta, la macchina da scrivere sono contenitori virtuali di linguaggio, sono linguaggio non ancora attualizzato, e devono la loro esistenza all'interno dell'azienda a niente altro all'infuori della loro capacitá intrinseca di produrre linguaggio.

Nella sua forma tradizionale, il linguaggio si realizza normalmente in oggetti materiali definitivi, come documenti, relazioni, cartelle informative eccetera. Tali oggetti presentano almeno i seguenti inconvenienti:

1- sono difficilmente correggibili -

2- sono accessibili ognuno ad una sola persona alla volta -

3- per essere "passati" da una persona ad un'altra, devono attraversare uno spazio fisico -

4- sono reperibili con una difficoltá che cresce rapidamente in funzione del loro numero; il tenerli in ordine é una fonte, e spesso non irrilevante, di altro lavoro -

5- occupano uno spazio fisico ingente.

5.2 IL SALTO DI QUALITA'

Se si riflette sulla questione, non si fa fatica a comprendere come tutto ciò che è linguaggio possa essere immerso e trattenuto in un computer, anziché su carta. Il passaggio dalla memoria di un computer al supporto cartaceo non è, del resto, un problema. La cosa, per di più, presenta i vantaggi omologhi ma di segno inverso rispetto ai problemi visti in precedenza:

1- i documenti sono facilmente correggibili -

2- sono continuamente a disposizione di tutti coloro che li debbono consultare -

3- dal primo momento che il documento esiste, esso può essere reso disponibile dal suo estensore ad altre persone senza che nulla di fisico si sposti nell'azienda; ovvero, immediatamente.

4- esistono vari modi razionali di organizzare i documenti, e quindi di richiamarli con facilità. Si pensi, ad esempio, alla struttura ad albero del file **system** di UNIX.

5- l'esistenza di un nuovo documento non comporta alcun incremento dello spazio fisico occupato all'esterno del computer.

Nell'ambito di questa ottica generale, ci occuperemo di seguito di alcuni strumenti che UNIX mette a disposizione dell'utente rispetto a questa problematica.

Il primo problema, come si è visto, è lo scambio di informazioni. Sotto UNIX esistono alcuni comandi che consentono la gestione di posta elettronica tra i vari utenti. Il più semplice di tali comandi è **write**. Esso consente di entrare in comunicazione con un utente che sia attualmente nel sistema, e che non sia in ambiente editor. Se componiamo

```
write paolo
```

e il sistema non ci nega il permesso di scrittura, nel qual caso ovviamente ci avvisa con un messaggio, ció che da quel momento componiamo a terminale viene spedito all'utente paolo una riga per volta, ossia ad ogni [RETURN]. Per uscire dallo stato di comunicazione, battere un EOF.

Piú potente é il comando **mail** , il quale consente di mandare messaggi o file ad un utente anche non collegato. Quando quel-l'utente userá di nuovo il sistema, potrá ricevere la posta attraverso lo stesso comando.

Per quanto la posta elettronica possa essere importante, lo strumento su cui si accentra molta dell'attivitá di office automation é l'editor.

5.3 GLI EDITOR

Il concetto di **editor** é di fondamentale importanza nella pratica odierna dell'informatica. Tale strumento é divenuto ultimamente via via piú duttile e piú diffuso, al punto che rappresenta ormai una delle prime cose con cui il principiante, su qualsiasi sistema, prende confidenza.

L'editor é un possibile ambiente di lavoro; un ambiente diverso da quello cosí detto **dei comandi** , ovvero quello in cui la macchina propone un **prompt** e aspetta il nome di un programma da eseguire. In editor, viceversa, noi ci muoviamo come all'interno di un file; é soprattutto in questo ambiente, quindi, che possiamo scrivere e correggere. Ad esempio, una volta che sia stato concepito e scritto su carta un programma, si deve "andare in editor" per "metterlo dentro", ossia per renderlo disponibile all'elaboratore, e dunque eseguibile.

Sotto UNIX esistono vari editor, piú o meno diffusi. Spesso accade che, nello stesso sistema, siano presenti piú editor; si tenga presente, infatti, che un editor, come qualsiasi altra funzione che un sistema ci metta a disposizione, non é altro che un programma; e la presenza di un certo programma non esclude la presenza di un altro programma.

Tra tutti gli editor che girano sotto UNIX, noi abbiamo deciso di illustrare brevemente il **vi** e l' **ined.** Per il primo va

detto che, essendo stato inglobato nel SYSTEM III, puó essere considerato, in questo momento, il più diffuso, e il più vicino ad essere considerato uno standard. Per quanto riguarda il secondo, esso ha dalla sua due importanti fattori: i) essendo della **Interactive** , ed avendo l' **IBM** adottato quella versione di UNIX, é quello destinato a diffondersi maggiormente in ambiente IBM, a cominciare dal PC; ii) é senza dubbio il più facile da usare, e forse il più potente.

5.4 L'EDITOR VI

Il nome **vi** , cui si é fatto cenno nell'Introduzione, viene da visual In ambiente **vi** si ha infatti che il file su cui si lavora viene visualizzato a pieno schermo; tuttavia si deve fare riferimento alle righe quando si danno i comandi. Avvertiamo qui che, per il seguito, i caratteri o gruppi di caratteri racchiusi tra parentesi quadre non sono da comporre, ma corrispondono direttamente a un tasto della tastiera.

5.4.1 ENTRARE IN EDITOR

Per entrare in ambiente editor il comando é **vi** , seguito eventualmente dal nome di un file:

vi testo.

A questo punto, se il file esiste giá esso viene, per quanto possibile, visualizzato, mentre se il file prima non esisteva lo schermo s'imbianca, presentando solo il segno ~ a sinistra di ogni riga.

Nell'ultima riga in basso si leggono il nome del file e la sua attuale estensione. Dando il comando

[CTRL] g

nella stessa riga compariranno invece informazioni sulla posizione attiva, ossia su cui siamo posizionati. In ogni caso l'ultima riga non appartiene al file, ma è riservata ai messaggi del sistema.

5.4.2 LO STATO DI INSERIMENTO

Il modo più ovvio per cominciare ad usare un editor è quello di passare in stato di **inserimento** ; questo si ottiene semplicemente con il comando

i

Si esce invece dallo stato di inserimento con il tasto

[ESC].

In fase di inserimento ci si comporta come con una normale macchina da scrivere: si compone il testo riga dopo riga, andando a capo con l'apposito tasto di ritorno carrello, normalmente

[RETURN].

Il modo migliore per comprendere bene un editor è ovviamente quello di usarlo: se avete a disposizione un terminale, e siete un utente accreditato di un sistema UNIX, create in **vi** un file, e scrivete la prima pagina di questo libro. Se commettete un errore, e ve ne accorgete quando siete ancora sulla stessa riga, potete tornare indietro con il tasto [BACK SPACE], oppure con [CTRL] w, che permette di tornare indietro di una parola intera. Se l'errore è in una riga precedente, per il momento ignoratelo; presto ci occuperemo in dettaglio di come effettuare le modifiche.

5.4.3 USCIRE DALL'EDITOR

Se siamo in stato di inserimento, per prima cosa dobbiamo chiudere questa fase con il tasto [ESC]. Se non lo facessimo, qualsiasi comando verrebbe interpretato come semplice testo, e inserito nel file, anziché eseguito.

A questo punto possiamo uscire dall'editor con il comando

:wq

Cosí facendo, otteniamo di passare alla riga dei comandi (con i due punti), di scrivere su disco il nostro file (w - da write) e di uscire dall' editor (q - da quit). Infatti il lavoro di editing non avviene mai direttamente sul file su disco, ma su una copia temporanea. Con il comando che abbiamo visto il file temporaneo viene ad essere copiato su quello definitivo. Lo stesso effetto si puó ottenere con il comando ZZ.

Se viceversa vogliamo uscire senza "salvare" ció che é avvenuto nella sessione di editor, ossia vogliamo annullare d'un colpo tutto quanto fatto, modifiche e inserimenti, dovremo uscire con il comando

:q!

nel qual caso rimane in essere solo ció che era stato salvato dall'ultimo comando :w. Infatti, come abbiamo visto, le due lettere che compongono il modo di uscire normale (salvando su disco) rappresentano anche separatamente un comando. Cosí con

:w

possiamo salvare continuando a lavorare in editor; é da notare che tale comando accetta come opzione il nome di un file, che possiamo quindi creare in modo che sia conforme a quello su cui stiamo lavorando. Il comando

:q

consente l'uscita dall'editor, ma viene accettato soltanto se non sono state eseguite modifiche. Aggiungendo il punto esclamativo, come abbiamo visto, ossia dando

:q!

il comando viene comunque eseguito, e si ottiene l'effetto di annullare la sessione di editing.

Queste possibilità risultano molto comode quando si vuole duplicare un file, apportandovi però qualche lieve modifica. Supponiamo di avere un file di tipo programma. Noi vogliamo creare un nuovo programma, assai simile, ma con qualche istruzione diversa, conservando tuttavia anche il primo. Possiamo allora entrare in editor del primo:

vi prog1.b

quindi effettuare le modifiche, poi dare il comando

:w prog2.b

In questo modo avremo creato il nuovo programma richiesto, di nome **prog2.b.** Infine, possiamo dare il comando

:q!

in modo tale che **prog1.b** rimanga inalterato.

5.4.4 COME CORREGGERE

Per usare in modo efficace un editor in fase di correzione, la prima cosa da imparare bene è quella di spostarsi velocemente lungo il file. In ambiente **vi** , i tasti direzionali per spostare il cursore non sono rappresentati dalle solite frecce, ma dalle quattro lettere

l, h, j, k.

Per un verso, fin che non ci si è fatta l'abitudine, questo modo di spostarsi appare meno comodo, in quanto meno "intuitivo", rispetto al ricorso ai tasti con le frecce; va tuttavia considerato che questi comandi di spostamento, come del resto quasi tutti i comandi di **vi** , ammettono una specificazione

numerica, attraverso un numero preposto, senza blank in mezzo, alla lettera selezionata. Cosí

12j

fará scendere il cursore di 12 righe. Per tutti i comandi, dunque, si é sottinteso, ma reso modificabile, il parametro 1 di default.

Sono inoltre comodi i seguenti comandi di spostamento:

[CTRL] f	per avanzare di una pagina
[CTRL] b	per andare una pagina indietro
[CTRL] d	per avanzare di 12 righe
[CTRL] u	per andare indietro di 12 righe

Per ricordarsi meglio, si noti che

f	abbrevia	"forward"
b	abbrevia	"backward"
d	abbrevia	"down"
u	abbrevia	"up"

Inoltre, con il comando **nG** , dove n é un numero, ci si porta alla riga n-esima. Non premettendo alcun numero, si va alla fine del file.

Altri utili comandi si riferiscono allo schermo. Con

H

e con

L

ci si sposta, rispettivamente, alla prima e all'ultima riga della videata. Come al solito, possiamo premettere al comando un parametro numerico: con **3H** , ad esempio, andremo alla terza riga dall'alto. Con il comando **M** , inceve, ci si porta all'inizio della riga a metá dello schermo.

Inoltre noi possiamo agire sullo schermo anziché sul file. Con il comando

z [RETURN]

la linea su cui stiamo operando diventa la prima della schermata; con

z-

il controllo si posiziona alla fine, e con

z.

si posiziona al centro.

Infine, possiamo fare riferimento alle parole, "cercando" in avanti o indietro nell'ambito del file. Con la barra, o slash, (/) , diviene attiva l'ultima riga del video, e possiamo scrivere una stringa, la quale verrá cercata in avanti. Il cursore si fermerá alla prima occorrenza della stringa. Dando poi **n** , si passa alla prossima occorrenza. Se al posto della barra diamo il punto interrogativo, la ricerca viene effettuata all'indietro anziché in avanti; analogamente, dando **N** al posto di **n** , si trovano le successive occorrenze all'indietro.

Possiamo anche specificare che intendiamo cercare le occorrenze della stringa solo quando esse occupano una certa posizione nella riga. Cosí

^ significa a inizio riga

$ significa a fine riga

Inoltre, particolarmente utile é il carattere speciale punto (.). Esso assume in ambiente **vi** lo stesso significato che ha il punto interrogativo in ambiente SHELL, ovvero sta per quasiasi carattere. Cosí

/e.a

ci fará trovare tutte le stringhe in cui compaiono una **e** , poi un carattere qualsiasi, poi una **a** , come

```
catena
   ---

era
---

esagono
---
```

Una ulteriore possibilitá molto utile é quella di specificare un **range** di caratteri che ci interessano fornendo il primo e l'ultimo collegati da un trattino, il tutto racchiuso da quadre:

[a-z]

significa quindi "qualsiasi lettera minuscola".

5.4.5 SPOSTARSI ALL'INTERNO DELLA RIGA

Quando ci siamo posizionati sulla riga che ci interessa, possiamo ovviamente spostarci in essa con il comando di cursore a destra o a sinistra (l e h). Esistono tuttavia diverse altre possibilitá che rendono piú agevoli gli spostamenti:

^ va a inizio riga.

¦ va a inizio riga, ma accetta un parametro numerico: n¦ va al carattere ennesimo.

$ va a fine riga.

Altri comandi fanno invece riferimento alle parole, ovvero interpretano il blank come separatore. Cosí

e

va alla fine della parola su cui siamo posizionati, mentre

w

e

b

ci portano rispettivamente alla prossima parola e alla precedente. E' da notare che questi tre ultimi tengono conto dei segni di punteggiatura; volendo ignorarli, basta dare gli stessi caratteri, ma maiuscoli.

5.4.6 I COMANDI DI CANCELLAZIONE

Per cancellare, possiamo far uso di comandi che si riferiscano a uno o più caratteri, a una o più parole, a una o più righe. Il comando

x

cancella un carattere alla volta (salvo premettergli un parametro). Si riferiscono invece a un'intera parola i comandi

dw

e

db

l'uno agendo sulla parola che segue, l'altro che precede il cursore.

Se invece vogliamo cancellare tutta la porzione di riga che segue il cursore, il comando é

D

Noi possiamo anche agire su tutte le parole che seguono la posizione del cursore fino a un certo carattere escluso o compreso. Nei due casi rispettivamente avremo

dt{c}

e

df{c}

dove c é il carattere a cui facciamo riferimento.

Possiamo invece riferirci a un'intera riga, o a più righe, premettendo un numero, con il comando

dd

Ancora più potenti sono i comandi

dL

e

dG

i quali cancellano tutte le righe fino alla fine dello schermo il primo, fino alla fine del file il secondo.

5.4.7 LE SOSTITUZIONI

Una volta che abbiamo cancellato una parte di testo, spesso accade che sia opportuno aprire dello spazio per inserire. Allo scopo si ricordi che con

o

si apre una riga bianca sotto quella attuale, e si passa in stato di inserimento. Lo stesso avviene con

O

con la differenza che la riga viene aperta sopra anziché sotto quella attuale.

Un modo per operare sostituzioni è dunque quello di sopprimere la parte di file che non è adeguata, e quindi inserire la parte nuova. Vi sono tuttavia, ovviamente, comandi che consentono di fare le due cose contemporaneamente.

Con il comando

nr{c}

rimpiazzeremo con **n** caratteri c i primi **n** caratteri. Il comando ha quindi senso o per sostituire un carattere ad un

altro, oppure perché vogliamo ottenere **n** caratteri uguali, caso questo, ovviamente, non molto frequente. Di maggiore utilitá é invece

s

Con esso si ottiene la cancellazione del carattere corrente, e il contemporaneo passaggio in stato di inserimento. (Si ricordi che si esce dallo stato di inserimento con [ESC]).

Poiché, come al solito, possiamo premettere un numero al comando, otteniamo l'effetto di eliminare **n** caratteri, e quindi poterli sostituire immettendo direttamente ció che vogliamo loro sostituire, e di potere poi continuare ad inserire altre cose, senza ulteriori soppressioni.

Possiamo poi riferirci alle parole anziché ai caratteri con il comando

cw

il quale, analogamente ad **s** , fa sí che la parola corrente venga sovrascritta, e che si passi poi in stato inserimento.

In analogia ai comandi **dt** e **df** visti in precedenza, abbiamo

ct{c}

e

cf{c}

i quali agiscono dal punto attuale fino al carattere selezionato escluso o compreso. Infine il comando

cc

sostituisce una riga intera.

E' utile menzionare a questo punto i comandi per annullare i cambiamenti fatti. Con

U

possiamo riportare la riga su cui é stato fatto l'ultimo cambiamento allo stato precedente alle modifiche. Se vogliamo

agire su piú righe, annullando in ogni caso l'effetto del-
l'ultimo comando dato, ricorreremo a

u .

5.4.8 DUPLICARE O SPOSTARE RIGHE

Come abbiamo visto, possiamo lavorare su righe intere. Il co-
mando

Y

copia una riga in memoria, senza asportarla dal file. Se di
seguito diamo il comando

p

la riga copiata viene riprodotta nella posizione attuale. Sup-
poniamo di stare mettendo dentro un programma, e di arrivare a
un gruppo di istruzioni molto simili ad istruzioni precedenti,
che quindi abbiamo giá inserito. Ci posizioniamo allora sulla
prima delle istruzioni che ci interessano. Supponiamo che esse
siano dieci. Diamo poi, nell'ordine, i seguenti comandi

1) 10Y
2) G
3) p

Con il passo 1) otteniamo che venga fatta una copia in un buf-
fer di memoria delle dieci righe seguenti; con 2) ci spostiamo
alla fine del file; infine, con 3), copiamo dal buffer al fi-
le, e quindi riproduciamo le dieci istruzioni.

Un caso la cui opportunitá é altrettanto frequente é quello di
voler spostare una parte di un file da un punto ad un altro,
ovvero sopprimendola nel punto da cui la si preleva. Il proce-
dimento per ottenere questo risultato é del tutto analogo a

quello appena visto: basterá sostituire al comando **10Y** il comando **10dd** :

1) 10dd
2) G
3) p

Infatti anche ció che viene cancellato con il comando **dd** viene automaticamente salvato, come se avessimo dato il comando **Y**

Il meccanismo di scrittura delle cancellazioni nel buffer é piú precisamente il seguente. Il sistema memorizza le ultime nove cancellazioni effettuate, in nove aree diverse. La piú recente é nella prima, la piú remota nell'ultima. Se vogliamo riprendere una cancellazione fatta tre volte fa, dovremo premettere **"3** al comando **p** , e dare

"3p

Ad ogni successiva cancellazione, il contenuto viene ad occupare l'area piú prossima, ossia la prima, quella cui accediamo direttamente con il comando **p** , mentre le cancellazioni piú vecchie si spostano di una posizione, e solo la piú remota va perduta.

5.4.9 AGIRE SU TUTTO IL FILE

Abbiamo visto come intervenire su un numero qualunque di righe. Possiamo usufruire di comandi ancora piú potenti posizionandoci sulla riga dei comandi (la penultima dello schermo) con il carattere speciale

:

Un comando dato di seguito avrá effetto per tutto il file, o per quella parte di file che noi stessi specifichiamo. Infatti ad ogni comando possiamo premettere la riga di partenza e di fine dell'effetto, fornendo due numeri separati dalla virgola. In merito si ricordi che il semplice punto si riferisce alla riga attuale, e il carattere speciale **$** si riferisce alla fine

del file.

Il formato generale di un comando sará allora

{prima riga},{ultima riga}comando{argomenti} .

Ad esempio, se vogliamo sostituire la parola **files** con **file** dal punto attuale in poi daremo il comando

.,$,s/files/file/g

dove il punto fa riferimento alla posizione attuale, e la g finale fa sí che la sostituzione avvenga anche per le occcrrenze che non sono la prima della riga.

5.5 L'EDITOR INED

Per entrare in ambiente **ined** si deve dare il comando

e {nome-file} .

Qualora il file non esista giá, il sistema invita a premere il tasto [USE]; questo provoca l'effetto di **creare** il file, e renderlo direttamente disponibile nella finestra di editor.

Si puó poi scrivere in esso come se si avesse a che fare con una macchina da scrivere con visore. I tasti con le freccette consentono di dirigere il cursore dove si vuole.

L'editor ined fa largo uso di funzioni associate direttamente ai tasti. Questo privilegia, dal punto di vista pratico, l'uso di terminali provvisti di tasti funzionali. La corrispondenza tasti/funzioni deve essere stabilita in fase di installazione. Da questo punto di vista possiamo dire che il vi é meno dipendente dal terminale, e quindi piú portabile, ma l'ined, come vedremo, risulta piú comodo da usare.

Quando si commette un errore di dattilografia, o si deve cambiare, per una qualsiasi ragione, una parte di testo giá composta, é sufficiente riportarsi sulla parte sbagliata e sovrapporre ad essa il testo corretto. Per spostarsi facilmente e velocemente lungo il testo, si hanno a disposizione alcuni tasti speciali. Essi sono:

[GO TO]	che riporta immediatamente il controllo a inizio file -
[ENTER] [GO TO]	che riporta immediatamente il controllo a fine file -
[ENTER] {n} [GO TO]	che posiziona il controllo alla n-sima riga del file -
[HOME]	che posiziona il controllo all'inizio della finestra di editor -
TASTI DI SPOSTAMENTO	rappresentanti quattro frecce orientate. Spostano il punto attualmente attivo, su cui si posizionera' il prossimo carattere inviato da tastiera, nelle rispettive posizioni -
[+PAGE] [-PAGE]	che spostano il testo in avanti o indietro di una pagina di video (21 righe) -
[+LINE] [-LINE]	che spostano il testo in avanti o indietro di un terzo di una pagina di video (7 righe).

Con questi tasti diventa estremamente comodo portarsi ad un punto prefissato del testo. Per correggere, abbiamo a disposizione due metodi: a) quello di sovrascrivere il testo vecchio; b) quello di passare in INSERTMODE, e inserire aggiunte che spostano il resto della riga verso destra, lasciandolo inalterato. Per passare in INSERTMODE basta premere il tasto corrispondente della tastiera. Al di sotto della finestra comparirá la scritta

***** I N S E R T M O D E *****

Per ritornare alla situazione normale, basta premere lo stesso tasto una seconda volta.

Per cancellare caratteri sbagliati, recuperandone lo spazio, ovvero senza sovrascrivere nulla, esiste il tasto

[DEL] ,

che elimina un carattere alla volta, e ricompatta il testo facendo slittare la parte della riga che segue il carattere cancellato da destra a sinistra.

Infine, per ricercare una stringa, ci si deve comportare nel modo seguente:

1- battere [ENTER]

2- comporre la stringa richiesta

3- battere [+SRCH] per cercare più avanti nel testo e [-SRCH] per cercare nell'ambito del testo precedente.

Se si vuole poi passare alla successiva occorrenza della stessa stringa, basta compiere nuovamente la sola operazione del punto 3, in quanto il sistema ricorda l'ultima stringa cercata.

5.5.1 PIU' POTENTI COMANDI DI EDITOR

Esistono altri tasti di particolare utilità per lavorare in editor. Essi consentono di spostare, duplicare, cancellare intere righe o intere pagine. Essi sono:

[OPEN]

[CLOSE]

[RESTOR]

[PICK]

[PUT]

Il tasto [OPEN] inserisce una riga bianca prima della riga attuale del cursore. Il tasto [CLOSE], viceversa, cancella la riga attuale. Il tasto [RESTOR] riscrive l'ultima riga che è stata cancellata con il [CLOSE]. Usando congiuntamente questi ultimi due tasti, è quindi possibile prelevare una riga dal punto del testo in cui risiede, e porla in un altro punto qualsiasi.

Analogamente, [PICK] e [PUT] vengono usati, normalmente, in modo congiunto: il primo copia una riga, ma non la fa scomparire come il [CLOSE]; il secondo la scrive nella posizione voluta. L'effetto congiunto di [PICK] e [PUT] non è quindi quello di spostare una riga di testo da una parte a un'altra, ma quello di ricopiare una riga, lasciandola sopravvivere anche nel punto da cui si è prelevata.

Questi tasti consentono quindi di lavorare riga a riga, anziché carattere a carattere. Ma, più ancora, essi accettano un parametro numerico, che dica loro di agire su un numero qualsiasi di righe. Tale parametro va passato battendo [ENTER] e di seguito il numero voluto, e impostando poi il tasto funzionale che si intende invocare. Se, ad esempio, vogliamo copiare le successive sette righe di testo e riproporle in fondo al file, dovremo eseguire i seguenti passi:

1- battere [ENTER]

2- battere [7]

3- battere [PICK]

4- battere [ENTER]

5- battere [GO TO]

6- battere [PUT]

Lo stesso discorso vale per tutti gli altri tasti visti in questo paragrafo. Analogamente, anziché un parametro numerico, possiamo passare ad essi le coordinate della zona di video interessata, battendo [ENTER] e poi spostando il cursore a nostro piacimento, e infine impostando il tasto voluto.

5.5.2 COME DEFINIRE UN'AREA PER IL CURSORE

E' possibile definire un'area del cursore per inserire (ossia per il tasto [OPEN]), cancellare (per il tasto [CLOSE]), o per ricopiare (per il tasto [PICK]). Definire un'area di cursore consta dei seguenti tre momenti:

1 - Posizionare il cursore all'inizio dell'area che si deve definire e premere il tasto [ENTER]; nella posizione del cursore appare allora il segno #, e compare la scritta <ENTER> a sinistra sotto la finestra.

2 - Definire l'area con i tasti di spostamento del cursore. Quando, dopo avere premuto [ENTER], si premono i tasti di posizionamento del cursore, a sinistra sotto la finestra appare il commento: <***CURSOR DEFINED***>. E' possibile in questo modo definire per il cursore una o più parole, una o più righe, una o più colonne. Se ci si vuole riferire a più righe intere, conviene ricorrere al comando di spostamento verticale del cursore, mentre se ci si vuole riferire soltanto ad alcune parole consecutive di una stessa riga bisogna usare il tasto che sposta il cursore verso destra.

3 - Premere [OPEN], [CLOSE] o [PICK]. Il cursore tornerá allora alla sua posizione di partenza.

Il tasto [INS] serve dunque per aggiungere le parole nell'ambito di una riga; il tasto [OPEN] aggiunge invece un'intera riga per volta. Per fornire lo spazio per aggiunte piû grandi di una riga, basta battere ripetutamente il tasto [OPEN]. E' consigliabile eseguire questa operazione quando si prevede di fare aggiunte consistenti, per scongiurare il pericolo di scrivere sopra il testo preesistente, cancellandolo. Dopo che si sono eseguite le aggiunte del caso, se rimangono righe bianche, si possono eliminare senza problemi con il tasto [CLOSE].

Per rendere piû veloce la procedura di aprire uno spazio notevole per le aggiunte che si intendono fare, si puó usare appunto il metodo di definire un'area di cursore. Per definire un'area per aggiunte, procedere nel modo seguente:

1 - Posizionare il cursore nel punto dove si vuole aprire l'area, e battere poi il tasto [ENTER];

2 - Usando il tasto che sposta il cursore verso il basso, spostarsi per tutto lo spazio che si vuole. Apparirá in basso la scritta <***CURSOR DEFINED***>;

3 - Premere il tasto [OPEN]. Si apre cosí lo spazio richiesto, il cursore ritorna allora alla sua posizione di partenza, e si puó procedere a comporre le aggiunte.

Per spostare una parte di testo da un punto a un altro, é possibile allo stesso modo definire un'area di cursore, analogamente a quello che si fa per il tasto [OPEN]:

1 - Si deve innanzitutto posizionare il cursore all'inizio dell'area che si vuole cancellare, e premere [ENTER];

2 - Si muova poi il cursore verso il basso per tutto il paragrafo che si intende cancellare, inclusa la riga bianca che lo separa dal paragrafo successivo;

3 - Si prema infine il tasto [CLOSE].

Tutta la parte interessata scompare dallo schermo, ed è in effetti eliminata dal file; essa può essere ripristinata con il tasto [RESTOR].

5.5.3 LAVORARE SU PIU' FILE

E' possibile, mantenendo attiva la finestra attuale su un file, aprire una seconda finestra di editor su un altro file. E' necessario posizionarsi sul bordo di sinistra o su quello in alto della finestra, quindi

1- battere [ENTER] -

2- battere il nome del file su cui si vuole aprire la seconda finestra -

3- battere il tasto [WINDOW] .

A questo punto avremo due finestre contemporaneamente attive su due file diversi. Il video verrà suddiviso orizzontalmente se ci eravano posizionati sul margine di sinistra, verticalmente se ci eravamo posizionati su quello in alto. Possiamo passare dall'una all'altra con il tasto [CHWIN]. Possiamo eliminare la seconda finestra aperta con

[ENTER] [WINDOW]

Vale la pena di notare che possiamo impostare i tasti [CLOSE] o [PICK], quindi cambiare finestra con [CHWIN], e poi premere [RESTOR] o [PUT] rispettivamente, e collocare così l'esito dell'operazione nell'altro file. In questo modo è facile prelevare da documenti precedenti parti analoghe e inserirle nei nuovi.

Va notato infine che il procedimento di aprire ulteriori finestre è iterabile fino a un massimo di dieci finestre.

5.5.4 USCIRE DALL'EDITOR

Per uscire da una sessione di editor basta battere il tasto [EXIT]. Allora scomparirá la finestra, e ricomparirá il segnale di "prompt", $, che ci avvisa di essere ritornati al livello dei comandi.

Dando ora il comando

e

non seguito da alcun argomento, ritorneremo in editor dello stesso file, nel preciso punto in cui lo abbiamo lasciato.

Per uscire dalla sessione di editor senza salvare le correzioni fatte, ovvero annullando tutte le modifiche fatte, basta dare il comando

[ENTER] a [EXIT]

CAPITOLO VI

IL LINGUAGGIO C

6.1 LA FUNZIONE **main**.

Un programma in C è scritto normalmente in minuscolo, in formato libero, a differenza ad esempio del FORTRAN e del COBOL. I vincoli sintattici principali sono i seguenti:

a- Il programma non contiene il proprio nome all'interno, come avviene invece ad esempio in COBOL; il nome che gli viene attribuito deve necessariamente terminare con

.c

b- Ogni programma deve comprendere la funzione

main

La parte relativa a **main** è racchiusa tra parentesi graffe.

c- Il punto e virgola è il segno con cui termina uno statement, e non un separatore, come ad esempio in PASCAL.

d- Una funzione viene chiamata semplicemente citandone il nome, seguito dagli eventuali argomenti racchiusi tra parentesi tonde; qualora non vi siano argomenti, le parentesi tonde sono comunque obbligatorie (si ricordi che **main** è una funzione).

Il compilatore del C ha nome

cc

Per compilare un programma occorre dunque dare l'istruzione

```
cc nome.c
```

Il codice così creato ha nome

```
a.out
```

che é anche l'istruzione per effettuarne l'esecuzione.

In un programma in C, come in ogni linguaggio, si fa uso di funzioni; queste possono essere definite all'interno del programma stesso o appartenere alle librerie precostituite. Ad esempio,

```
printf
```

é una funzione di libreria che formatta e fornisce il suo output su video, salvo esplicita istruzione in contrario.

Siamo ora in grado di scrivere e **far girare** il nostro primo programma in c.

```
main() {
     printf("Questo e' il primo programma in C");
}
```

```
+--------------------------------------------------------------+
:       Si crei in editor il programma precedente.             :
:                                                              :
:       Lo si compili                                          :
:                                                              :
:       Lo si esegua                                           :
+--------------------------------------------------------------+
```

Si noti che:

a- main é una funzione con nessun argomento; la qual cosa é indicata da ().

b- Le {} racchiudono gli statements della funzione. Esse rappresentano l'analogo del BEGIN-END del PASCAL.

c- Il formato di uno statement é privo di vincoli, eccetto la presenza del ; finale.

6.2 VARIABILI E TIPI DI DATI

Consideriamo ora i tipi fondamentali di variabili del C. Come in ogni linguaggio di programmazione, anche in C sono presenti i più consueti tipi di dati:

interi

alfanumerici

reali

Anche in C i tipi delle variabili vanno dichiarati. Per convenzione ciò viene fatto all'inizio. Le istruzioni di dichiarazione sono:

int	per gli interi (16 bit)
long	per gli interi (32 bit)
char	per un carattere di un byte (8 bit)
float	per variabile a virgola mobile a singola precisione (32 bit)
double	per variabile a virgola mobile a doppia precisione (64 bit)

Il numero dei bit occupati è ovviamente dipendente dalla macchina; i valori qui forniti si riferiscono a sistemi a 16 bit. Da quanto sopra, si evince che i valori estremi su queste macchine sono

+(-)10 ** +(-)38

Per quanto riguarda il tipo **char** , i caratteri fanno riferimento al codice ASCII, secondo la seguente tabella:

0	^@	0	22	^V	26	44	,	54	66	B	102	88	X	130	110	n	156
1	^A	1	23	^W	27	45	-	55	67	C	103	89	Y	131	111	o	157
2	^B	2	24	^X	30	46	.	56	68	D	104	90	Z	132	112	p	160
3	^C	3	25	^Y	31	47	/	57	69	E	105	91	[	133	113	q	161
4	^D	4	26	^Z	32	48	0	60	70	F	106	92	\	134	114	r	162
5	^E	5	27	^[	33	49	1	61	71	G	107	93	]	135	115	s	163
6	^F	6	28	^\	34	50	2	62	72	H	110	94	^	136	116	t	164
7	^G	7	29	^]	35	51	3	63	73	I	111	95	_	137	117	u	165
8	^H	10	30	^^	36	52	4	64	74	J	112	96	`	140	118	v	166
9	^I	11	31	^_	37	53	5	65	75	K	113	97	a	141	119	w	167
10	^J	12	32		40	54	6	66	76	L	114	98	b	142	120	x	170
11	^K	13	33	!	41	55	7	67	77	M	115	99	c	143	121	y	171
12	^L	14	34	"	42	56	8	70	78	N	116	100	d	144	122	z	172
13	^M	15	35	#	43	57	9	71	79	O	117	101	e	145	123	{	173
14	^N	16	36	$	44	58	:	72	80	P	120	102	f	146	124	¦	174
15	^O	17	37	%	45	59	;	73	81	Q	121	103	g	147	125	}	175
16	^P	20	38	&	46	60	<	74	82	R	122	104	h	150	126	~	176
17	^Q	21	39	'	47	61	=	75	83	S	123	105	i	151	127	del	177
18	^R	22	40	(	50	62	>	76	84	T	124	106	j	152			
19	^S	23	41	)	51	63	?	77	85	U	125	107	k	153			
20	^T	24	42	*	52	64	@	100	86	V	126	108	l	154			
21	^U	25	43	+	53	65	A	101	87	W	127	109	m	155			

(usare il comando **code** per verificare questa tabella)

Una dichiarazione puó essere fatta in modo multiplo per più variabili, separate dalla virgola. Come ogni statement, essa deve terminare con il punto e virgola:

int a, b, c;

I nomi di variabili sono costruiti liberamente con i caratteri

A-Z, a-z, 0-9 e _

con l'unico vincolo che il primo carattere non sia numerico. E' buona norma tuttavia usare uniformemente il maiuscolo o il minuscolo.

Poiché i simboli degli operatori aritmetici sono i soliti dei linguaggi evoluti (COBOL, FORTRAN, PL/I, PASCAL ...), ossia

+ - * /

per le operazioni razionali, siamo in grado ora di scrivere un programma un po' più significativo del primo; il seguente moltiplica tre numeri e stampa a video il risultato:

```
main() {
     int a, b, c, per;
     a = 2; b = 7; c = 4;
     per = a * b * c;
     printf("Il risultato e' %d", per);
}
```

Il simbolo **%d** ha lo scopo di far fornire il risultato in base decimale. Ce ne occuperemo più in dettaglio. E' da rilevare che il formato degli statements è del tutto libero: è lecito porre molti statements su di una stessa riga, o spezzarne uno su più righe. Come unica limitazione, non è consentito spezzare un nome o un operatore.

```
+----------------------------------------------------------------------+
:        Scrivere un programma che sommi alcuni numeri e stampi        :
:        il risultato.                                                 :
:                                                                      :
:        Lo si compili                                                 :
:                                                                      :
:        Lo si esegua                                                  :
:                                                                      :
:        Si spezzino gli statement in vario modo, controllando         :
:        cosa succede.                                                 :
+----------------------------------------------------------------------+
```

Per quanto il linguaggio C sia decisamente libero quanto a regole di formattamento, esiste uno standard di scrittura a cui normalmente ci si attiene. Tale standard può essere ot-

tenuto automaticamente dando in input il sorgente al comando cb :

```
cb < brutto.c > bello.c
```

6.3 INPUT/OUTPUT

Le funzioni fondamentali per la gestione dell'input/output sono

getchar

e

putchar

oltre alla giá vista **printf**.

La funzione **getchar** prende un byte alla volta dallo standard input, e assume quel byte come valore. Cosí l'istruzione

```
a = getchar();
```

assegna alla variabile **a** il valore della funzione **getchar** , ovvero il carattere letto. Quando la funzione raggiunge la fine del file, essa rende un EOF. EOF é un intero il cui valore, dipendente dalla macchina, é scritto nel file di libreria

stdio.h

La funzione **putchar** , al contrario, rende allo standard output un carattere per volta. Si consideri l'esempio di programma seguente:

```
#include <stdio.h>
main() {
        int a;
        a = getchar();
        putchar(a);
}
```

La prima riga del programma causa la lettura del file **stdio.h.** Viene poi dichiarata intera la variabile **a.** Si noti che essa non puó essere dichiarata char, perché EOF é un intero. Viene poi letto un byte dallo standard input, e reso allo standard output. Normalmente, quindi, viene prelevato un carattere da tastiera, e reso a video.

6.4 ALCUNI SEGNI SPECIALI

Il linguaggio C é spesso usato per programmazione di sistema; come é noto, UNIX é prevalentemente scritto in C. Cosí in questo linguaggio hanno particolare rilievo i numeri in base otto. Una sequenza di cifre che cominci con zero é in C un numero in base otto, in cui quindi non possono comparire le cifre **8** e **9**. Per richiedere la notazione decimale di un numero, si ricorre al simbolo speciale

%d

come si é visto. L'effetto contrario, ovvero quello di presentare un numerale in base otto, con soppressione degli zeri iniziali, si ottiene con il simbolo

%o

dove l' o viene da **octal.**

Consideriamo il programma seguente:

```
main() {
	int n;
	n = 658;
	printf("Qual e' il valore di %d in base 8?", n);
	printf("%d in base 10 e' %o in base 8?", n,n);
}
```

Altri simboli speciali di frequente uso nella programmazione in C sono i seguenti:

\n	a capo
\t	tab
\b	backspace
\0	carattere nullo
\\	\

In particolare é utile **\n** , in relazione alla funzione **printf** Infatti é possibile gestire in questo modo l'inizio di una nuova riga.

Opzioni di formattamento importanti per la funzione **printf** , analoghe a **%o** , sono:

%c

per stampare un singolo carattere e

%s

per stampare un'intera stringa.

```
+-----------------------------------------------------------------+
|       Si riscriva il programma precedente facendo in modo che  |
|       la risposta alla domanda sia presentata su una nuova     |
|       riga.                                                     |
+-----------------------------------------------------------------+
```

6.5 SEGNI RELAZIONALI E OPERATORI BOOLEANI

Come si é detto, gli operatori aritmetici in C non differiscono dagli analoghi degli altri linguaggi evoluti. Ad essi va aggiunto l'operatore

%

il quale fornisce il resto di una divisione tra interi, e corrisponde al MOD del PASCAL:

```
x = a%b
```

vale il resto della divisione di a per b.

Il C non ha un tipo di dato booleano. I dati booleani vengono assimilati agli interi. Questo significa che il valore di una funzione é considerato vero tutte le volte che esso é diverso da zero.

Anche i segni relazionali non differiscono molto dai soliti, con l'eccezione dell'uguaglianza booleana, che si realizza replicando il segno di uguale. In C infatti il semplice segno di = é una assegnazione. Cosí, per testare l'uguaglianza di due variabili, a e b, si dovrá scrivere

```
if(a == b)
```

Si considerino attentamente le seguenti tabelle riassuntive, in cui gli operatori del C sono messi in parallelo con quelli del COBOL:

OPERATORI ARITMETICI					
COBOL	+	-	*	/	
C	+	-	*	/	%

OPERATORI RELAZIONALI					
COBOL	=	<>	<>	<=	>=
C	==	!=	<>	<=	>=

<table>
<tr><th>OPERATORI BOOLEANI</th><th></th><th></th><th></th></tr>
<tr><td>COBOL</td><td>NOT</td><td>AND</td><td>OR</td></tr>
<tr><td>C</td><td>!</td><td>&&</td><td>||</td></tr>
</table>

Va tenuto presente che in C si esistono molti livelli di priorità tra gli operatori. Questo consente spesso l'omissione delle parentesi, e, in generale, una scrittura più compatta.

Nella seguente tabella é riportato l'ordine di priorità per principali operatori.

OPERATORI DEL LINGUAGGIO C

Operatore	Associativita'
() [] -> .	a destra
! ++ -- * & sizeof	a sinistra
/ %	a destra
+ -	a destra
<< >>	a destra
< <= > >=	a destra
== !=	a destra
&	a destra
^	a destra
¦	a destra
&&	a destra
¦¦	a destra
? :	a sinistra
= += -=	a sinistra
,	a destra

6.6 ESPRESSIONI CONDIZIONALI

Come in quasi tutti i linguaggi, una condizione si esprime in C con un **if.** La sintassi é

```
if(espressione) statement else statement
```

dove **else statement** é opzionale. In merito va notato che:

a- l'espressione da testare deve essere racchiusa tra parentesi;

b- il valore aritmetico zero é considerato falso, ogni altro valore vero;

c- a differenza del PASCAL o del BASIC, non ci vuole la parola **then.**

d- dopo ogni statement che non sia seguito da una parentesi tonda chiusa ci vuole ;.

e- come di consueto, **else** si lega all' **if** piú recente.

Una peculiaritá del C, che rende agevole l'uso dell' **if** , é che é semplice raggruppare un numero quanto si vuole grande di statements in dipendenza della condizione semplicemente racchiudendoli tra parentesi graffe.

Inoltre, é ammessa la seguente scrittura:

```
espressione1 ? espressione2 : espressione3;
```

La sua semantica é la seguente: essa vale **espressione2** quando **espressione1** é vera; **espressione3** altrimenti. Cosí, scrivendo

```
x = ( a > b ? a : b);
```

si assegnerá ad x il valore maggiore tra a e b.

Si osservino le seguenti istruzioni.

```
if(a < b) {
        x = a;
        a = b;
        b = x;
}
```

Completarle in modo che si abbia un programma in grado di girare.

6.7 LA RIPETIZIONE

La ripetizione, o **loop** , si ottiene in C fondamentalmente in tre modi: con l'istruzione **while** , con l'istruzione **for** e con l'istruzione **do ... while.**

La sintassi e la semantica dell'istruzione **while** ricordano da vicino il WHILE ... DO del PASCAL. La forma generale é:

```
while(espressione) statement
```

Di fronte a un **while** il sistema reagisce nel modo seguente:

a) valuta l'espressione racchiusa tra parentesi;

b) se essa é vera (come per l' **if** una espressione é comunque vera quando il suo valore é diverso da zero), esegue **statement** e torna ad a).

Come si vede, poiché il test precede l'esecuzione dello statement, un **while** puó eseguire **statement** zero volte.

Si consideri l'esempio seguente:

```
#include <stdio.h>
main() {
        int a;
        while ((a = getchar()) != EOF)
                putchar(a);
        }
```

La quarta riga mostra un tratto tipico del C: si esegue un'assegnazione,

a = getchar()

e, nell'ambito della stessa istruzione, si controlla poi se la variabile così assegnata é diversa da EOF. Il motivo per il quale é possibile in C procedere in questo modo é che una assegnazione é considerata come un operatore. Quindi essa, oltre ad attribuire un valore ad una variabile, ha inoltre un suo valore intrinseco, precisamente il valore della variabile a sinistra di =. che puó essere usato in qualsiasi espressione. Questo fa sí che si possa scrivere

x = y = z = 7 ,

attribuendo il valore 7 contemporaneamente a x, y e z.

```
+-------------------------------------------------------------+
:       Che cosa fa il programma ora visto?                   :
:                                                             :
:       Lo si riscriva senza far ricorso all'istruzione di    :
:       assegnazione e test contemporanei.                    :
+-------------------------------------------------------------+
```

Va notato che l'operatore relazionale != precede il segno di assegnazione = nella scala delle prioritá. Questo rende necessarie le parentesi che racchiudono

a = getchar() ,

senza le quali la variabile a assumerebbe valore 0 o 1 a seconda che getchar() fosse diverso o uguale a EOF.

Il **for** del C é sostanzialmente una generalizzazione del **while**. Una istruzione dipendente da **for** in definitiva non é altro che un modo di porre un **while** assieme alla inizializzazione e all'incremento, oltre naturalmente alla condizione di fine della ripetizione. La sintassi del **for** é la seguente:

```
for (inizializzazione; espressione; incremento)
                  statement
```

Lo sviluppo semantico é:

```
        inizializzazione;
        while (espressione) {
            statement
            incremento;
        }
```

Si noti che l'ordine dei tre elementi di **for** é fisso: al primo posto va messa l'inizializzazione, la quale viene eseguita comunque una sola volta; il secondo elemento entro parentesi é l'espressione che deve essere vera perché la ripetizione continui; la terza posizione é riservata infine all'incremento, che viene eseguito comunque dopo l'esecuzione dello statement. Si noti ancora che l'incremento, a differenza degli altri due elementi, non é seguito dal punto e virgola.

Consideriamo l'esempio seguente:

```
main() {
        int i, n, a;
        a = 1;
        n = 7;
        for (i = 1; i <= n; ++i)
                a * = i;
        printf("Il fattoriale di 7 e' %d",i)
}
```

Il programma mostra un esempio tipico dell'uso del **for.** Dopo la parole **for** , entro parentesi, si trovano tre elementi. Il primo,

```
i = 1
```

inizializza la variabile intera i a 1. Il secondo,

```
i <= n
```

stabilisce che il loop di ripetizione deve essere continuato

fin tanto che i é minore o uguale a n. Infine, il terzo elemento,

```
++i
```

rappresenta in C l'incremento: esso é l'analogo di

```
i = i + 1
```

come si scriverebbe negli altri linguaggi.

Si noti che, in C, una variabile viene inizializzata a zero se non compare una istruzione in contrario. Se un loop comincia da zero, l'inizializzazione puó essere omessa, ma non cosí il punto e virgola, che é obbligatorio. Lo stesso discorso vale anche per l'espressione condizione di continuazione: essa puó essere omessa purché si conservi il punto e virgola. In questo caso essa é assunta come sempre vera. Anche l'incremento puó essere omesso (vedremo piú avanti come si puó uscire da un loop privo di incremento o di condizione di fine), ma, poiché esso é l'ultimo elemento della terna entro parentesi, non ci vuole qui il punto e virgola.

Nell'esempio precedente va notata la scrittura

```
++i ,
```

modo tipico del C di effettuare l'incremento. Il C ammette tanto la scrittura

```
i++
```

quanto quella

```
++i
```

oltre, ovviamente, alle analoghe

```
i--
```

e

```
--i
```

La differenza sta nel fatto che, in una istruzione complessa,

l'incremento segnato a sinistra della variabile viene eseguito prima, quello a destra dopo l' uso della variabile.

Così, se abbiamo

```
int x, y, z;
      y = 6;
```

dopo l'esecuzione dell' istruzione

```
x = y++
```

y vale 7, ma x vale 6. Viceversa se abbiamo l'istruzione

```
z = ++y
```

y vale ugualmente 7, e lo stesso valore ha anche z. Infatti nel primo caso y viene incrementato dopo l'esecuzione dell'assegnazione, mentre in

```
z = ++y
```

prima si pone y = 6 + 1, e poi si pone z = y.

Questa scrittura per l'incremento è più efficace, in quanto in quasi tutti i calcolatori è presente direttamente una istruzione di macchina che esegue l'incremento di una variabile. Una volta di più il C tradisce la sua origine di linguaggio di sistema.

```
+-------------------------------------------------------------------+
:        Si crei in editor il programma precedente.                 :
:                                                                   :
:        Lo si modifichi, ponendo n = 10.                           :
:                                                                   :
:        Che cosa succede?                                          :
+-------------------------------------------------------------------+
```

Il tipo int ha valore massimo 2 ** 15.

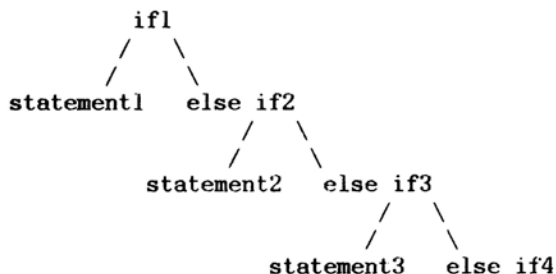
```
+-------------------------------------------------------------------+
:        Si crei in editor il programma seguente:                   :
:                                                                   :
:        #include <stdio.h>                                         :
:        main(){                                                    :
:               int a,n;                                            :
:               n = 0;                                              :
:               while((a = getchar()) != EOF)                       :
:                    if(a == '\n')                                  :
:                           ++n;                                    :
:               printf("%d righe\n", n);                            :
:        }                                                          :
:                                                                   :
:        Si compili il programma precedente.                        :
:                                                                   :
:        Si provi a dare il comando                                 :
:                                                                   :
:               ls : a.out                                          :
:                                                                   :
:        Si interpreti il risultato.                                :
:                                                                   :
:        Si scriva un programma che conti i caratteri di un         :
:        file.                                                      :
:                                                                   :
:        Lo si esegua.                                              :
+-------------------------------------------------------------------+
```

L'istruzione **do ... while** ha un coomportamento sintattico e semantico assai simile al loop realizzato con il **while.** La differenza sta nel fatto che in essa il test viene eseguito in coda anziché in testa al loop. Avviene allora che lo statement venga comunque eseguito almeno una volta.

6.8 L'ISTRUZIONE **switch**.

Supponiamo di avere, ad un certo punto di un programma, un certo numero di situazioni mutualmente escludentisi le quali devono essere trattate diversamente. Questa eventualità si affronta di solito con una serie di **if** nidificati:

```
if(a == b)
        statement1;
else
        if(a == c)
                statement2;
        else
                if(a == d)
                        statement3;
                else
                        printf("Valore fuori limite");
```

In pratica si compiono tante scelte binarie l'una dentro al-l'altra, secondo lo schema concettuale seguente:

Il C consente una scrittura compatta per un caso di questo genere. L'istruzione adatta é quella di **switch** , che ha il formato seguente:

```
switch (a) {
        case b :
                statement1;
                break;
        case c :
                statement2;
                break;
        case d :
                statement3;
                break;
        default:
                printf("Valore fuori limite");
                break;
        }
```

La presenza di **default** é opzionale: essa fa sí che un certo statement venga eseguito quando nessuno degli altri casi si verifica.

E' da rilevare che l'struzione **break** causa l'uscita immediata dallo **switch.** In assenza di **break** il programma esegue tutte le successive istruzioni dentro lo **switch** , anche se il caso preposto a sinistra non si verifica. Questo perché il **case** costituisce una label e non un **if.** Dunque l'istruzione di switch senza alcun **break** all'interno realizza lo stesso costrutto del GO TO DEPENDING ON del COBOL.

L'istruzione **break** ha lo stesso effetto anche se posta all'interno di un **while** o di un **for** , ovvero causa l'uscita immediata dal loop.

All' interno di un loop si puó usare l'istruzione **continue** : essa fa partire immediatamente la successiva iterazione del loop. Quindi essa porta il controllo all'incremento se immersa in un for e alla condizione se immersa in un **while.**

6.9 I VETTORI

In C, come in quasi tutti i linguaggi, é possibile costruire dei vettori i cui elementi siano di un tipo di dati fondamentale. Il caso piú ovvio é quello di dichiarare un vettore di interi:

```
int a [5];
```

la precedente dichiarazione introduce un vettore di nome a di interi con indice da 0 a 4. Infatti in C la prima posizione é indiciata con 0, e non con 1 come al solito. Se dunque un vettore ha n elementi, l'indice massimo introdotto é n - 1. Va rilevato che le parentesi quadre sono riservate ai vettori, e ne racchiudono quindi il relativo numero di indici, cosí come le parentesi tonde sono devolute a racchiudere gli argomenti di una funzione. Ogni inversione di uso é illegale.

Come in altri linguaggi, il concetto di vettore si puó generalizzare in quello di matrice, ovvero di vettore a piú di una dimensione. Una matrice a due dimensioni si dichiarerá

```
int a [5] [30];
```

dove il primo numero indica le righe e il secondo le colonne.

6.10 STRINGHE COME VETTORI DI CARATTERI

Una stringa é considerata un vettore di caratteri, nel linguaggio C cosí come sotto UNIX in generale. Una stringa di 20 caratteri sará dichiarata dunque nel modo seguente:

```
char riga [21];
```

La terminazione di una stringa é determinata dal carattere nullo, \0. E' opportuno porre tale carattere alla fine di un vettore di tipo stringa, perché molte funzioni e molti programmi giá esistenti lo richiedono. Quando qualcosa é racchiuso tra virgolette, il compilatore pone automaticamente il carattere \0 alla fine, senza che lo si debba istruire esplicitamente.

Consideriamo il programma seguente:

```
main() {
        int n;
        char riga [100];
        n = 0;
        while( (riga [n++] = getchar() ) != '\n');
        riga [n] = '\0';
        printf("%d;%s", n, riga);
}
```

```
+------------------------------------------------------------------+
:        Si comprenda ogni passo del programma.                    :
:                                                                  :
:        Si noti l'incremento fatto direttamente sull'indice       :
:        del vettore.                                              :
+------------------------------------------------------------------+
```

6.11 I PUNTATORI

In un linguaggio in cui una stringa è pensata come vettore di caratteri necessariamente riveste grande importanza il concetto di puntatore. Come al solito, un puntatore è l'indirizzo di un qualche cosa, e come al solito esso rappresenta un modo veloce ed efficiente di pervenire a quel qualche cosa. Il che corrisponde concettualmente a passare da un indirizzo al suo contenuto. L'indirizzo di un ente, qualora esista, è reso disponibile in C dall'operatore

```
&
```

Se, essendo a e b due interi, poniamo

```
a = &b;
```

noi otteniamo l'indirizzo di b scrivendo

```
&b
```

e lo assegnamo alla variabile a. L'effetto è quello di rendere a un puntatore a b.

Perché a sia usato dal C come un puntatore e non come una normale variabile, dobbiamo dichiararlo come tale. Questo si fa premettendo un asterisco alla variabile nell'ambito della solita dichiarazione. Così, prima della assegnazione appena vista avremmo dovuto dichiarare

```
int *a , b;
```

Se ora noi scriviamo

```
c = *a
```

il C é istruito ad usare a come indirizzo e non come contenuto. Il risultato sará quello di porre in c il contenuto di b, a cui abbiamo fatto accesso attraverso il puntatore a.

In sintesi, si ricordi che, se x ha un indirizzo, *& premesso ad x non é altro che x stesso:

x e' uguale a *&x

L'uso piú perspicuo dei puntatori é ovviamente quello di gestire in modo efficiente un vettore. Supponiamo di avere dichiarato la stringa

```
char a [20];
```

Per accedervi con un puntatore, dovremo dichiarare anche

```
char *x;
```

A questo punto abbiamo determinato l'uso come puntatore di x. Se ora poniamo

```
x = &a [17];
```

il nostro puntatore conterrá l'indirizzo del diciottesimo carattere di a. Se ora incrementiamo x

```
x++;
```

otterremo che x punti al carattere successivo, ossia il diciannovesimo. (Si noti che il nome del vettore da solo é assunto come l'indirizzo del primo, ossia a equivale all'indirizzo a [0]).

```
+------------------------------------------------------------+
|        Scrivere la funzione                                |
|                                                            |
|                             length                         |
|                                                            |
|        che conta il numero di caratteri di una stringa,    |
|                                                            |
|        prima senza far uso di puntatori e poi con essi.    |
+------------------------------------------------------------+
```

Il ricorso ai puntatori ci consente di scrivere una efficiente funzione che copia una stringa a in una stringa b:

```
strcopy(a, b)
        char *a , *b; {
        while(*b++ = *a++);
}
```

Si noti che la dichiarazione char *a e quella char a [] sono equivalenti. Un puntatore puó infatti essere pensato come un vettore di cui non é specificata la dimensione.

6.12 PUNTATORI A MATRICI

Una importante distinzione, il cui fraintendimento puó essere fonte di errori per il neofita del C, é quella tra una matrice e un vettore di puntatori. Supponiamo di creare un vettore di puntatori, ciascuno dei quali punti ad un vettore. E' proprio questo che succede quando si vuole accedere velocemente ad un certo numero di stringhe. La dichiarazione relativa a questa situazione prende la forma

```
int *a [10]
```

Ogni elemento di a é quindi un puntatore, che indirizza ad un vettore. La situazione é ben diversa nel caso di una dichiarazione del tipo

```
char a [10] [10]
```

In questo secondo caso abbiamo una matrice, ovvero un vettore a due dimensioni, costituito di 100 celle. In entrambi i casi é sensato il riferimento

```
a [7] [7]
```

e questo puó confondere le idee. Per afferrare la differenza, si pensi tuttavia al fatto che la dichiarazione di una matrice puó gestire esclusivamente elementi della stessa lunghezza, mentre un vettore di puntatori a vettori puó essere tale che ogni elemento indirizzi a vettori dalla lunghezza piú diversa.

6.13 ARGOMENTI DI UN PROGRAMMA

Quando un programma viene mandato in esecuzione, gli vengono passati due argomenti, convenzionalmente chiamati

argc

e

argv

Il primo di essi non é altro che il contatore dei parametri con cui é stato invocato il comando. Il secondo, **argv** , é un puntatore a un vettore di stringhe di caratteri che contiene gli argomenti, uno per stringa. Per convenzione, argv [0] é il nome con cui il programma é stato chiamato. Il contatore di argomenti, **argc** , vale di conseguenza almeno 1. Quando questo si verifica, il comando é stato chiamato senza argomenti.

```
+--------------------------------------------------------------+
:        Scrivere un programma che fornisca in output gli       :
:        argomenti che gli vengono passati, compreso il nome   :
:        del programma stesso                                   :
+--------------------------------------------------------------+
```

Confrontare il programma precedente con questa possibile soluzione:

```
main(argc, argv)  /* questo programma stampa i suoi argomenti */
int argc;
char *argv [];
{
        while (--argc > 0)
              printf("%s%c", *++argv, (argc>1) ? ' ' : '\n');
}
```

Poiché **argv** é un puntatore al punto iniziale del vettore delle stringhe che rappresentano gli argomenti, incrementandolo di 1 con l'istruzione ++argv si fa sí che esso punti ad argv [1] anziché ad argv [0]. Ogni ulteriore incremento lo sposta sul successivo argomento, e *argv punta a quell'argomento. Nel

contempo, viene decrementato argc. La condizione di fine é argc uguale a zero.

L'esempio precedente mostra anche come si possono inserire i commenti in un programma.

In C, come in generale sotto UNIX, é prassi comune denotare premettendo il segno - quegli argomenti che introducono flags o parametri opzionali.

E' buona norma scrivere i programmi in modo tale che analizzino gli argomenti e i parametri opzionali indipendentemente dall'ordine nel quale sono forniti. Tale convenzione é adottata dai programmi giá presenti nel sistema.

6.14 STRUTTURE DI DATI

In C é possibile organizzare dati logicamente correlati in strutture. La forma della dichiarazione é la seguente:

```
struct {
        char  nome [20];
        int   anni;
        char  sesso;
} persona;
```

Questa dichiarazione definisce **persona** come una struttura, i cui membri sono **nome** , **anni** e **sesso.** In seguito possiamo fare riferimento a un membro della struttura con la forma

nome-struttura.membro

Ad esempio, possiamo correttamente eseguire le seguenti assegnazioni:

```
persona.anni = 34
persona.sesso = 'm'
```

Naturalmente un concreto vantaggio si ottiene non tanto nello strutturare i dati in questo modo, ma nel potere trattare questi tipi di dati come un tutto unico, creando ad esempio vettori di strutture. Supponiamo di dovere considerare 100 indi-

vidui, i cui attributi siano quelli dell'esempio precedente. Potremo dichiarare, anziché

```
struct {
        char  nome [20] [100];
        int   anni [100];
        char  sesso [100];
} persona;
```

sinteticamente

```
struct {
        char  nome [20];
        int   anni;
        char  sesso;
} persona [100];
```

Noi possiamo usare i puntatori anche in riferimento a strutture. Consideriamo l'esempio seguente:

```
struct {
        char  nome [20];
        int   anni;
        char  sesso;
} persona [100], *punt;

punt = &persona [0];     /* o anche  punt = persona */
```

Questo fa sì che **punt** punti alla struttura, e lo inizializza come indirizzo del primo elemento di **persona.**

Il modo di fare riferimento a un membro di una struttura attraverso un puntatore é

```
p -> membro-struttura
```

Il simbolo -> sta a significare il fatto che p punta a un membro di una struttura, ed é usato solo in questo contesto.

6.15 FUNZIONI

Abbiamo fin qui fatto riferimento, in modo più o meno esplicito, ad alcune funzioni, e al modo di farle intervenire in un programma. Vediamo ora di affrontare l'argomento in modo più sistematico. Come si é detto, una funzione puó essere di libreria, e quindi va semplicemente invocata, come eventuale passaggio di argomenti, oppure essere definita all'interno del programma stesso. E' questo secondo, evidentemente, il caso che ci interessa più da vicino.

L'esigenza di introdurre una funzione deriva dalla pratica della buona programmazione: quando, nell'ambito di un programma, si puó isolare un particolare problema, che puó essere visto come parte a se stante, é buona norma affrontarlo in una routine a parte, che viene invocata dalla funzione principale, e ad essa rende il controllo. E' questo approccio concettuale che consente la programmazione top-down.

Una funzione sará dunque una parte di un programma, la quale ne riproduce a sua volta lo schema sintattico generale: si ricordi che **main** puó essere considerata una funzione. Cosí essa avrá i suoi argomenti, le sue dichiarazioni, e una parte operativa racchiusa tra graffe. Ecco una funzione che calcola il maggiore tra due interi:

```
max (a, b)
        int a, b; {
        int x;
        x = (a > b ? a : b);
}
```

In una funzione puó essere inserito lo statement

return

che provoca il ritorno alla routine chiamante. Esso é opzionale, poiché quando la funzione ha esaurito la sua esecuzione, tale ritorno avviene comunque; il **return** diventa quindi inutile se posto alla fine di una funzione. Ma tale statement é molto utile soprattutto perché puó essere seguito da una qualsiasi espressione, che viene cosí passata alla funzione chiamante.

Possiamo riscrivere la funzione ora vista in modo più efficace:

```
max (a, b)
        int a, b; {
        return(a > b ? a : b);
}
```

In questo modo otteniamo che **max** rende al flusso chiamante a nel caso che esso sia il maggiore, b altrimenti.

Il modo con cui il programma e la funzione si passano variabili va considerato attentamente. Le variabili sono passate in C secondo la **call by value** ; ció significa che alla funzione chiamata é passata una copia dei suoi argomenti, dei quali essa non conosce l'indirizzo. Essa non puó quindi in nessun modo modificare il valore di alcuno dei suoi inputs, in quanto opera su copie locali. Questo, a differenza di quanto avviene in altri linguaggi, ad esempio in FORTRAN, in cui si ha invece la **call by name** , per il quale la variabile é consegnata in tutto e per tutto alla funzione chiamata.

In questo modo in C non sarebbe possibile da parte delle funzioni modificare il valore di un argomento, anche quando desiderato. La soluzione da adottare é quella di passare alla funzione non la variabile ma il **puntatore** ad essa. L' esempio mostra una funzione che scambia gli argomenti in ingresso. Essa é chiamata nel modo seguente:

```
swap ( &a, &b)
```

Il corpo della funzione é il seguente:

```
swap (i,j)
        int *i, *j; {

        int temp;
        temp=*i;
        *i=*j;
        *j=temp;
}
```

Un altro modo di affrontare questo tipo di problema é quello di fare riferimento a variabili **esterne** o **globali** , conosciute da tutte le parti del programma, e quindi da tutte le funzioni, per il loro nome, anziché a variabili **locali** , di ambito determinato.

6.16 VARIABILI LOCALI E GLOBALI

Una variabile dichiarata all'interno di una funzione compare e scompare con essa e va inizializzata ogni volta, poiché non si conserva tra una chiamata e la successiva. Di più, due variabili interne a funzioni diverse, anche se definite con lo stesso nome, non hanno niente a che spartire l'una con l'altra. In un programma di questa forma

```
min () {
        int a;
}
max() {
        int a;
}
```

la variabile a della funzione **min** e la variabile a della funzione **max** non avrebbero alcuna relazione tra di loro.

Noi possiamo tuttavia renedere una variabile **globale** , ovvero renderla disponibile a tutto un programma, in modo che essa sia vista e trattata da tutte le funzioni.

Questo si ottiene inserendo la dichiarazione al di fuori di una funzione. La variabile così dichiarata, detta **globale** , è riconosciuta da tutte le funzioni che nel testo la seguono (così una variabile dichiarata prima del **main** , è nota a tutto il programma).

Una variabile globale dichiarata in un certo punto del testo puó essere resa disponibile anche alle funzioni che precedono la sua dichiarazione. Per fare questo dobbiamo dichiararla come **esterna** , con una dichiarazione apposita.

```
main () {
        int a, b;
}

min () {
        extern int a, b, c[];
}
max() {
        extern int a, b, c[];
}
int a, b, c[10];

\. . . altre funzioni
```

Naturalmente dovrá esistere da qualche parte una dichiarazione del tipo

```
int a, b, c[100];
```

Quando tale dichiarazione precede le funzioni che usano le variabili in questione, l'apposita specificazione che si realizza con la dichiarazione di **extern** diventa superflua. Tutte le parti del programma interessate vedono infatti una dichiarazione che sia a loro precedente. Il ricorso all' **extern** diventa altrimenti indispensabile, e fa sí che il compilatore acceda alla corretta definizione delle variabili.

6.17 FUNZIONI RICORSIVE

Una funzione in C puó chiamare se stessa. Questo significa che puó essere definita in modo ricorsivo.

In prima approssimazione possiamo dire che una funzione é definita ricorsivamente quando tale definizione

a - fornisce il modo di calcolare un primo valore;

b - fornisce il modo di passare dal valore di un argomento al valore dell'argomento successivo.

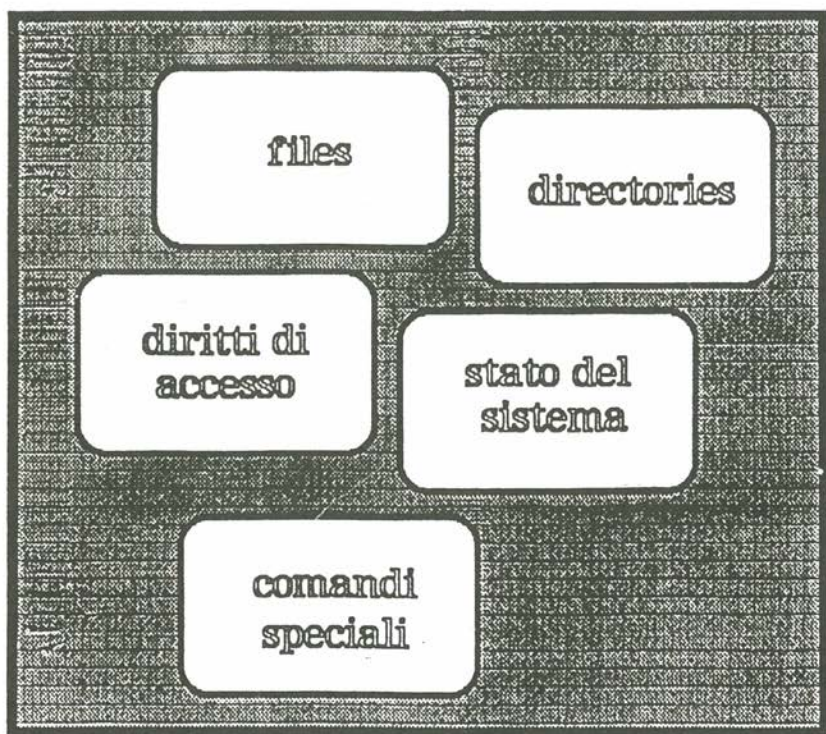
Cosí, ad esempio, possiamo definire ricorsivamente la funzione di **fattoriale** dicendo che

a - il fattoriale di 1 é 1;

b - il fattoriale di n é n * fattoriale (n - 1).

Dal momento che abbiamo un primo fattoriale (quello di 1 fornito da a -), e sappiamo, noto il fattoriale di n - 1 , calcolare quello di n (per b -), siamo in grado di calcolare il fattoriale di qualsiasi numero, o, detto altrimenti, ogni fattoriale resta individuato.

```
fattoriale (n)
        int n;  {
        return ( n <= 1 ? 1 : fattoriale (n-1) );
}
```

Se pur di sfuggita, non possiamo esimerci dal notare che la teoria della ricorsivitá ha acquistato una enorme importanza da quando per piú vie, sia di logica matematica che di teoria della computabilitá e degli algoritmi, si é giunti alla conclusione che le funzioni effettivamente calcolabili siano tutte e sole quelle ricorsive (c.d. **tesi di Church**).

i comandi di unix

APPENDICE

DESCRIZIONE DEI PRINCIPALI COMANDI DI UNIX

cat	*date*	*lpr*	*passwd*	*su*	*who*
cb	*df*	*ls*	*pcat*	*sync*	*write*
cc	*du*	*mkdir*	*pr*	*tail*	
cd	*echo*	*more*	*ps*	*tee*	
chgrp	*file*	*mv*	*pwd*	*test*	
chmod	*find*	*newgrp*	*rm*	*umask*	
chown	*grep*	*nice*	*rmdir*	*uniq*	
cmp	*ld*	*nl*	*sort*	*unpack*	
cp	*ln*	*pack*	*split*	*wc*	

cat [-u] [-s] [file...]

copia e concatenamento di file

DESCRIZIONE

cat legge i file specificati negli argomenti e li scrive sequenzialmente sullo standard output. Qualora non venga indicato alcun file, cat legge dallo standard input.

PRINCIPALI OPZIONI

-u i caratteri sono inviati sull'output file non appena sono letti dal comando; non viene effettuata alcuna memorizzazione temporanea in buffer.

-s non viene generato alcun messaggio di errore qualora i file specificati negli argomenti non esistano.

ESEMPI

cat primo secondo > terzo

Il contenuto del file terzo é costituito dalla concatenazione di primo e secondo.

cat miofile > /dev/tty4

Invia miofile al terminale tty4.

PRECAUZIONI

cat primo secondo > terzo

qualora il file terzo esista, il comando provoca la perdita del vecchio contenuto.

cat primo secondo > primo

non é permesso.

cb

migliora la scrittura di un programma scritto in C.

DESCRIZIONE

Serve ad adeguare i propri programmi agli standard di pretty writing dell'ambiente UNIX. Il comando legge dallo standard input e scrive sullo standard output. Cosí, se vogliamo applicarlo ad un programma giá scritto e salvato, dovremo redirigere l'input e l'output.

ESEMPI

cb < brutto.c > bello.c

Il programma brutto.c viene indentato e scritto sul programma bello.c.

cc [opzioni] file...

compilatore C

DESCRIZIONE

Il compilatore tratta i file trasmessi come parametri in modo dipendente dal suffisso appeso al loro nome:

.c il file é considerato un sorgente C

.o il file é considerato un modulo oggetto da trasmettere al loader

.s il file é considerato un sorgente in assembler

Quando non esplicitamente indicato, il compilatore manda in esecuzione il loader e produce un programma eseguibile.

PRINCIPALI OPZIONI

Alcune opzioni sono interpretate direttamente da cc. Altre sono invece trasmesse a ld (vedi).

-c sopprime la fase di caricamento e crea uno o più file oggetto con suffisso .o

-O attiva un passo di ottimizzazione del codice prodotto.

-S esegue solo il passo di compilazione, e lascia il risultato in un file assembler con suffisso .s

-P esegue solo il passo di preprocessing e lascia il risultato in un file con suffisso .i

-E esegue solo il passo di preprocessing e manda il risultato sullo standard output.

ESEMPI

```
cc main.c libreria.o
```

Compila il file main.c e produce un modulo eseguibile caricando anche il file oggetto libreria.o

```
cc prog.c -lx -o prog
```

Compila il file prog e risolvi i riferimenti cercando anche la libreria /lib/libx.a. Chiama prog il file eseguibile risultante.

```
cc -E prog.c >lista.c
```

Esegui il preprocessing del sorgente e manda il sorgente risultante sul file lista.c

cd [directory]

cambia il directory di lavoro

DESCRIZIONE

cd permette il posizionamento in un punto qualsiasi del file system. Qualora non sia specificato alcun argomento, cd riporta il directory corrente al directory di login oppure al directory specificato nella variabile HOME se questa é settata diversamente. L'argomento puó essere sia un path-name assoluto, quando il nome del directory destinazione comincia per / (radice del file system), che un path-name relativo, specificato a partire dal directory corrente. In UNIX SYSTEM V é possibile specificare in maniera abbreviata il path-name del directory destinazione facendo uso della variabile CDPATH. Se viene fornito come argomento un path-name parziale, qualora tale path-name non corrisponda ad un path-name relativo al directory corrente, il directory verrá ricercato a partire dai directory specificati nella variabile CDPATH.

ESEMPI

cd /usr/sergio

Il directory corrente diventa /usr/sergio. Il path del directory é stato specificato in maniera completa a partire dalla radice del file system.

cd lettere/ricevute

Il path del directory destinazione é specificato a partire dal directory corrente. Se prima della esecuzione del comando il directory corrente é /usr/sergio, dopo sará /usr/sergio/lettere/ricevute.

cd ..

Ci posiziona al livello immediatamente superiore della gerarchia dei directory. Ad esempio da /usr/sergio-/lettere a /usr/sergio.

cd lettere

Se lettere non è compreso nel directory corrente e la variabile CDPATH vale /usr/sergio:/usr/paolo il sistema cercherà prima il directory /usr/sergio/lettere. In caso negativo cercherà /usr/paolo/lettere.

chgrp gruppo file...

cambia gruppo di file e directory

DESCRIZIONE

chgrp cambia l'identificatore di gruppo di file e directory. L'operazione è consentita al proprietario dei file e directory specificati oppure al super-user. I gruppi consentiti sono quelli specificati nel file /etc/group. **ESEMPI**

chgrp altri *

Il gruppo proprietario dei files del directory corrente diventa altri.

chmod protezioni file...

cambia le protezioni di file e directory

DESCRIZIONE

chmod cambia il modo (le protezioni) dei file e directory specificati. Il modo può essere precisato in maniera simbolica o assoluta. Qualora specificato in maniera assoluta, il modo è un numero ottale di tre cifre, delle quali la prima specifica le protezioni per il proprietario del file, la seconda per il gruppo di appartenenza, la terza per il resto degli utenti. Ogni cifra viene costruita sommando 1 se si vuole il permesso di esecuzione, 2 se si vuole il permesso di scrittura, 4 il permesso di esecuzione. Se il modo è specificato in maniera simbolica, esso ha la seguente struttura:

[chi] op protezioni

chi puó essere un sottoinsieme di

    u per utente
    g per gruppo
    o per altri oppure
    a per tutti

op puó essere:

    + per aggiungere protezioni
    - per togliere protezioni
    = per assegnare precisamente le protezioni

protezioni é di solito un sottoinsieme di

    r per lettura
    s per scrittura
    x per esecuzione

Solo il proprietario di un file o directory ed il super-user possono modificarne le protezioni.

ESEMPI

chmod 640 lettera

il proprietario di lettera ha il permesso di lettura e scrittura del file (4 + 2 = 6), il gruppo associato il solo permesso di lettura, gli altri utenti nessun permesso.

chmod 777 miocomando

Il proprietario, gli utenti del gruppo associato al file e tutti i rimanenti utenti hanno il permesso di lettura, scrittura ed esecuzione di miocomando.

chmod ug+rw relazione

Aggiunge i permessi di lettura e scrittura per il proprietario ed il gruppo del proprietario del file relazione.

chmod +x miocomando

Rende eseguibile miocomando.

chown proprietario file...

cambia il proprietario di file

DESCRIZIONE

chown cambia il proprietario di file e directory. L'operazione é consentita al proprietario dei file e directory specificati oppure al super-user. Gli utenti del sistema sono specificati nel file /etc/passwd.

ESEMPI

chown stefano *

Rende stefano proprietario di tutti i file contenuti nel directory corrente.

cmp [-l] [-s] file1 file2

confronta due file

DESCRIZIONE

cmp confronta il contenuto dei due file specificati negli argomenti e segnala, qualora sia riscontrata una differenza, il numero del primo byte che differisce, ed il suo numero di linea.

OPZIONI PRINCIPALI

-l esamina le differenze tra i file e visualizza, per ogni byte diverso, il numero d'ordine ed il valore, in ottale, del byte nei due file.

-s non invia alcun messaggio all'utente ma ritorna un return code, che puó essere utilizzato in una procedura SHELL oppure in un programma applicativo.

ESEMPI

cmp comandol comandol.old

Fornisce un messaggio se i due file differiscono.

cmp - file2

Confronta lo standard input con il file file2.

cp file1 file2
cp file1 [file2...] directory
copia file

DESCRIZIONE

Nella prima forma specificata cp copia il file indicato nel primo argomento nel file indicato come secondo argomento. Nella seconda modalitá cp copia i file specificati inserendoli nel directory destinazione.

ESEMPI

cp letteral lettera2 lettera3 LETTERE

Copia i file letteral lettera2 lettera3 nel sottodirectory LETTERE.

cp /dev/tty miofile

Copia quello che viene battuto alla tastiera nel file miofile.

PRECAUZIONI

Qualora il file destinazione esista, il suo contenuto viene rimpiazzato.

date [MMddhhmm[yy]] ['+formato']

data ed ora

DESCRIZIONE

date visualizza la data e l'ora nel formato specificato. Permette al super-user di stabilire la data e l'ora corrente.

MM é il numero del mese
dd é il numero del giorno nel mese
hh é l'ora
mm é il minuto
yy é l'anno

Il formato é rappresentato da una stringa nella quale alcuni campi, costituiti dal carattere % e da un secondo carattere, sono rimpiazzati dal corrispondente valore quando la data é visualizzata.

%d é il giorno del mese
%m é il mese
%y é l'anno (ultime due cifre)
%H é l'ora
%M é il minuto
%S é il secondo
%n salta una riga

ESEMPI

```
date '+DATA: %d/%m/%y%nORA: %H-%M-%S'
```

Visualizza la data nel formato specificato, ad esempio:

```
DATA: 30/12/85
ORA: 16-15-10
```

df [-f] [-t] [file_system...]

visualizza lo spazio libero su disco

DESCRIZIONE

df determina il numero di blocchi liberi e di i-node disponibili nei file system enumerati tra gli argomenti oppure per tutti i file system montati sul sistema qualora l'argomento non venga specificato.

PRINCIPALI OPZIONI

-f viene soltanto riportato il numero dei blocchi liberi.

-t viene riportato, oltre al numero di blocchi liberi e di i-node disponibili, anche il numero totale di blocchi allocati per ogni file system.

ESEMPI

df -t

Visualizza il numero di blocchi liberi, i-node disponibili e blocchi allocati per ogni file system montato sul sistema.

df /dev/rp01

Visualizza il numero di blocchi liberi e i-node per /dev/rp01.

du [-ars] [nome...]

utilizzo dei dischi

DESCRIZIONE

du fornisce il numero di blocchi occupati per ogni file e directory elencato tra gli argomenti. Nel caso di directory vengono esaminati ricorsivamente anche tutti i sottodirectory. Se non viene specificato alcun argomento il comando agisce a partire dal directory corrente.

PRINCIPALI OPZIONI

-s viene riportato soltanto il numero totale di blocchi occupati dai directory specificati negli argomenti. Senza alcuna opzione viene invece riportato il totale per ogni directory e ricorsivamente anche per ogni sottodirectory.

-a viene riportato il numero di blocchi utilizzato da ogni file contenuto nei directory specificati negli argomenti e in tutti i loro sottodirectory.

-r genera messaggi di errore se qualche file o directory non puó essere esaminato. Ad esempio, viene inviato un messaggio di errore se devono essere percorsi directory protetti in esecuzione (ricerca).

ESEMPI

du -s /usr/giovanni /usr/stefano

Visualizza l'occupazione complessiva di memoria da parte dei file di giovanni e stefano.

du -a ¦ grep miofile

Fornisce il path-name del directory di miofile e lo spazio da esso occupato.

echo [-n] [argomenti]

scrive i suoi argomenti sullo standard output.

DESCRIZIONE

Gli argomenti vengono stampati sullo standard output separati da blank e con un newline in fondo. L'opzione -n serve ad eliminare tale newline. In questo modo si può ottenere un elenco di dati tutti su di una stessa riga.

ESEMPI

echo -n 'ls ' > file

In file vengono scritti, su una sola riga e separati da un solo blank, tutti i file del directory corrente.

file [-c] [-f file] [-m file] file ...

determina il tipo di file

DESCRIZIONE

file esamina il contenuto dei file specificati negli argomenti e cerca di determinarne il tipo, file eseguibile, sorgente c, directory... Talvolta per indovinare il tipo di un file, il comando ricerca alcuni byte speciali che "marcano" i file di un dato tipo, i "numeri magici". Ad esempio, i file eseguibili hanno in testa il byte 410 ottale.

ESEMPI

file * >miei_file

Determina il tipo dei file del directory corrente e redirige l'output sul file di nome miei_file.

find lista di pathname espressione
cerca file

DESCRIZIONE

Cerca ricorsivamente verso il basso nella struttura ad albero per ogni pathname specificato nella lista i file il cui nome soddisfi un'espressione regolare costituita combinando alcuni elementi attraverso parentesi, negazione (!), concatenazione o disgiunzione (-o); diamo di seguito alcuni di tali elementi.

-name nome

Si avvera per un file che si chiami come "nome";

-links n

Vero se il file ha n link;

-user nome-utente

Vero se il file appartiene a nome-utente;

-size n

Vero se il file é costituito da n blocchi;

-inum n

Vero se il file ha l'inode numero n;

-exec comando

Vero se l'esecuzione di comando restituisce valore 0;

-ok comando

Come il precedente, ma subordina l'esecuzione del comando alla risposta affermativa (y) data interattivamente;

-print

Sempre vero; serve a stampare il path-name corrente;

-newer file

Vero se il file processato é stato modificato più recentemente del file dato come argomento.

grep [opzioni] espr [file]

ricerca di stringhe ed espressioni regolari in un file

DESCRIZIONE

La famiglia di comandi a cui appartiene grep ricerca nei file specificati nella linea di comando (lo standard input se nessun file é specificato) le linee che soddisfano l'espressione regolare fornita come argomento (eventualmente una stringa) e, per default, le stampa sullo standard output. Se viene specificato più di un file prima della linea viene stampato il nome del file in cui é stata trovata.

Le principali opzioni riconosciute da grep sono:

-v stampa le sole linee che non contengono l'espressione.

-c viene stampato solo il numero di linee trovate..

-l stampa i soli nomi dei file che contengono linee che soddisfano l'espressione.

-n ogni linea stampata viene preceduta dal suo numero d'ordine nel file.

-b ogni linea stampata viene preceduta dal numero del blocco sul quale é stata trovata. (E' talvolta utile per determinare dal contesto i numeri dei blocchi su disco.)

-s sopprime i messaggi di errore dovuti a file mancanti o non leggibili dall'utente.

Occorre fare attenzione quando si usano i metacaratteri $, *, [, ^, |, (,) e nell'espressione poiché sono significativi anche per lo SHELL. Per non avere problemi la cosa migliore é mettere l'espressione tra apici singoli (').

ESEMPI

```
grep 'valerio' file.mio
```

Estrae tutte le righe che contengono l'espressione (in questo caso la stringa) valerio.

```
grep -n '[uU]nix' f.*
```

Estrae tutte le linee che contengono l'espressione regolare [uU]nix (che verrá espansa in unix oppure Unix) in tutti i file il cui nome inizia per "f." e le stampa facendole precedere dal nome del file in cui sono state trovate e dal relativo numero di linea.

Per ulteriori informazione si vedano anche i comandi egrep, fgrep, ed sed e sh.

ld [opzioni] file ...

link editor

DESCRIZIONE

ld accetta uno o più moduli oggetto e li combina in un programma eseguibile risolvendo i riferimenti esterni ai singoli file. ld cerca inoltre nelle librerie lib?.a (libF.a, libP.a, libc.a, libm.a, liby.a) residenti nei directory /lib e /usr/lib le routines Il modulo eseguibile prodotto ha il nome a.out come default.

PRINCIPALI OPZIONI

-s elimina dal file prodotto, per risparmiare spazio, la symbol table e le informazioni di rilocazione.

-lxx cerca i simboli esterni anche nella libreria di nome "libxx.a", dove xx è una stringa qualsiasi.

-n il file eseguibile ha la parte di testo read-only e condivisibile da tutti gli utenti che eseguono il programma.

-o xx il nome del programma eseguibile è xx invece di a.out.

ln file1 [file2]

crea un link

DESCRIZIONE

ln permette di avere riferimenti in diversi directory ad un medesimo file fisico. Uno stesso file può apparire in directory differenti con lo stesso nome oppure con nomi diversi. Il link ha lo stesso nome del file originale e viene inserito nel directory corrente. Se viene specificato il secondo argomento il link ha quel nome.

ESEMPI

```
ln /usr/stefano/lettera relazione
```

Crea nel directory corrente, ad esempio /usr/sergio, un riferimento al file /usr/stefano/lettera, chiamandolo relazione. Poichè esiste un solo file con due nomi distinti, ogni modifica a /usr/stefano/lettera è una modifica a /usr/sergio/relazione. Il comando rm elimina un link al file. La cancellazione fisica viene effettuata solo quando viene tolto l'ultimo link al file stesso.

lpr [-c] [-r] [-m] [-n] [file...]

spooler di stampa

DESCRIZIONE

lpr invia in una coda di stampa il contenuto dei file specificati negli argomenti. Se non viene indicato alcun file lpr stampa lo standard input.

OPZIONI PRINCIPALI

-m al termine della stampa viene inviato un messaggio all'utente tramite il servizio di mail.

-c effettua una copia dei file che vengono inviati in stampa.

-r cancella i file dopo averli stampati.

ESEMPI

```
lpr *.c
```

stampa i sorgenti c language.

ls [-logtasdrucifp] [nome...]

elenco dei file di un directory

DESCRIZIONE

ls elenca i file contenuti nei directory specificati. Se non viene specificato alcun nome fra gli argomenti ls elenca i file e directory contenuti nel directory corrente. Numerose informazioni possono essere determinate specificando le opzioni del comando.

PRINCIPALI OPZIONI

-l riporta per ogni file e directory una descrizione estesa. Vengono elencate le protezioni, il numero di links al file, l' utente ed il gruppo a cui il il file appartiene, la dimensione del file in byte e la data di ultima modifica del file stesso. Le protezioni sono riportate come un insieme di nove caratteri. Considerati a gruppi di tre, esprimono i permessi per il proprietario, il gruppo proprietario, ed il resto degli utenti. I tre caratteri comprendono una r (read) se é abilitata la lettura del file, w (write) se e permessa la modifica del file, x (execute) se é permesso eseguire il comando. Per un directory il carattere x significa permesso di ricerca di un file al suo interno. Ad esempio i caratteri rwxr-x--x potrebbero rappresentare l'insieme delle protezioni associate ad un comando che chiunque puó eseguire, il cui contenuto puó essere letto dal proprietario e dal gruppo proprietario, ma che solo l'utente proprietario puó modificare.
I nove caratteri relativi alle protezioni sono preceduti da un ulteriore carattere che specifica se il nome si riferisce ad un file ordinario, ad un directory o ad un dispositivo.

-p evidenzia i directory facendo seguire al nome il carattere /.

-a elenca anche i file e directory il cui nome comincia con . ad esempio il file .profile.

-u visualizza la data di ultimo accesso al file invece che la data di ultima modifica.

ESEMPI

ls -ap

Elenca tutti i file e sottodirectory del directory corrente, compresi i file il cui nome comincia per punto. Evidenzia i directory con il segno /.

mkdir directory...

crea un directory

DESCRIZIONE

mkdir crea i nuovi directory specificati negli argomenti. I directory sono creati con permesso di lettura, scrittura ed esecuzione per il proprietario, il gruppo ed il resto degli utenti. Le protezioni di default all'atto della creazione di directory e file possono essere modificate con il comando umask in UNIX SYSTEM V.

ESEMPI

mkdir lettere

Se il directory corrente è /usr/stefano, il comando crea, se l'utente ha il permesso di scrittura, il directory /usr/stefano/lettere.

more [-d] [-f] [-l] [-linea] [+/espr] file...

visualizza i file

DESCRIZIONE

more permette di visualizzare file una schermata alla volta. Ogni 22 righe di testo il comando arresta la visualizzazione e si pone in attesa di comandi. Numerosi argomenti possono essere definiti sulla linea di comando; specificando -linea la visualizzazione comincia dalla riga ennesima invece che dalla prima; specificando +/espressione_regolare, la visualizzazione comincia dalla prima riga contenente l'espressione regolare. Quando more é in attesa di comandi é possibile battere un

carriage return: verrá cosí visualizzata la linea successiva.

spazio: vengono visualizzate le 22 righe successive.

ni: vengono visualizzate le prossime n righe, n é un numero intero.

ns: salta n righe e visualizza la successiva schermata, n é un numero intero.

nf: salta n schermate e visualizza la successiva, n é un numero intero.

q: esce da more.

n/espr: cerca la ennesima espressione regolare, n é un numero intero.

=: visualizza il numero di linea corrente.

v: manda in esecuzione l'editor full screen vi a partire dalla linea corrente; quando si esce dall'editor si ritorna nel comando more.

h: visualizza informazioni di aiuto.

!comando: esegue il comando specificato nell'argomento.

.: riesegue l'ultimo comando.

i:n: visualizza l'iesimo successivo file, se sulla linea di comando, quando si è mandato in esecuzione il comando, sono stati specificati più file.

mv file1 file2

mv file1 [file2...] directory

sposta file e directory

DESCRIZIONE

Nella prima forma specificata mv muove il file indicato nel primo argomento nel file indicato nel secondo argomento. Il comando serve a cambiare il nome ad un file. Nella seconda modalitá mv sposta i file specificati inserendoli nel directory destinazione senza cambiarne il nome. La differenza fra mv e cp è che mv cancella il file sorgente lasciando al termine del comando una sola copia del file, il file destinazione, mentre cp lascia anche il file sorgente.

ESEMPI

mv LETTERA lettera

Rinomina il file LETTERA chiamandolo lettera.

PRECAUZIONI

mv file1 file2

Qualora file2 esista, il comando provoca la perdita del vecchio contenuto.

newgrp

cambia gruppo a un utente

DESCRIZIONE

Cambia l'identificazione del gruppo di chi lo invoca. Il directory corrente non viene cambiato, ma vengono modificate le protezioni per gli accessi ai file. Se il gruppo ha una password, e l'utente no, essa viene chiesta.

nice [-numero]
comando [argomenti]
rallenta l'esecuzione di un comando

DESCRIZIONE

Il comando nice consente di gestire la prioritá di scheduling di un comando.

Se si fornisce un numero preceduto dal trattino, tale numero viene sommato al numero che esprime la prioritá del comando (di default 10), fino a un massimo di 20. Si tenga presente che numeri piú alti esprimono una prioritá piú bassa.

Il super-user puó rendere piú veloce l'esecuzione dei propri comandi attribuendo loro un numero negativo (es. --5).

ESEMPI

nice -10 grep paolo *

Il comando "grep paolo *" viene eseguito con bassa prioritá.

nl [opzioni] file

filtro di numerazione delle linee

DESCRIZIONE

nl permette di numerare, con diverse opzioni, le righe del file specificato nell'argomento. Qualora.manchi il nome del file, nl numera le linee dello standart input. Il testo viene considerato spezzato in pagine logiche, ciascuna costituita da una intestazione, un corpo ed un fondo pagina. L'inizio di una intestazione é segnalata da una riga costituita dai soli caratteri ::: , l'inizio del corpo di ogni pagina da una linea costituita dai caratteri :: e l'inizio del fondo pagina da una linea con i caratteri : . Per default la numerazione riparte all'inizio di ogni pagina logica. E' possibile numerare tutte le righe della pagina logica, solo le righe dell'intestazione o del corpo o fondo pagina, le righe che contengono stringhe ricercate etc.

OPZIONI PRINCIPALI

-p non ricomincia la numerazione delle righe all'inizio di una nuova pagina.

-wnum num é un intero che specifica il numero di caratteri riservato per i numeri di linea.

-dxx xx sono due caratteri che consentono di modificare il valore di default per il separatore : e quindi del separatore di intestazione e corpo di pagina.

pack [-] file...

comprime file

pcat file...

visualizza file in formato compresso

unpack file...

espande file in formato compresso

DESCRIZIONE

pack esegue la compressione dei file specificati negli argomenti. L'ammontare della compressione effettuata dipende dalla distribuzione più o meno uniforme dei byte nel file da impaccare. pack cancella il file originario generando un file con lo stesso nome del file specificato nell'argomento e terminato dai due caratteri .z . Se viene specificato l'argomento -, pack fornisce informazioni quali il numero di volte in cui ogni byte presente nel file impaccato è usato. pcat consente di visualizzare e concatenare file compressi, analogamente a quanto cat fa per i file ordinari. unpack rigenera i file in formato esteso a partire dai corrispondenti file in formato compresso.

ESEMPI

pcat *.c

Comprime tutti i sorgenti c presenti nel directory corrente.

passwd

cambia la password di login

DESCRIZIONE

Ogni utente puó specificare una password che il sistema richiederá all'atto del login nel sistema. Una volta generata una password, essa é mantenuta in forma crittografata dal sistema nel file /etc/passwd. Anche se un utente ha accesso al file delle password, non puó leggere la password in chiaro.

ESEMPI

passwd bo347284

Crea una password o modifica quella giá esistente. Solo il super user puó cancellare una password.

PRECAUZIONI

Il super user ha il permesso di scrittura sul file /etc/passwd e quindi puó cancellare qualsiasi password creata dagli utenti.

pr [opzioni] [file...]

stampa impaginata di file

DESCRIZIONE

pr invia sullo standard output il contenuto dei file indicati negli argomenti oppure lo standard input in assenza di argomenti. Ogni pagina contiene una intestazione che riporta data e orario, nome del file corrente e numero di pagina.

PRINCIPALI OPZIONI

-p qualora l'output sia diretto al terminale, pr -p visualizza il contenuto dei file indicati una pagina alla volta, attendendo un carriage return prima di passare alla visualizzazione della pagina successiva.

-h "stringa" l'intestazione di default, che separa ogni pagina può essere modificata a piacere e posta eguale a stringa.

-n fa precedere ogni linea di testo dal numero progressivo di linea.

-wn l'opzione -w consente di specificare la larghezza della pagina di output. n é la larghezza indicata come numero di caratteri.

-lk l'opzione -l consente di specificare la lunghezza della pagina di output. l é la lunghezza indicata come numero di righe.

-k l'output é costituito da k colonne invece che una.

-d doppia spaziatura invece che singola tra le righe del testo.

-t non viene costruita alcuna intestazione di pagina e non viene lasciata alcuna riga bianca tra una pagina e l'altra.

ESEMPI

pr -p -n -t -123 file1

Visualizza il contenuto di file1, senza inserire intestazioni di pagina, fermandosi in attesa di un carriage return ogni 23 righe, numerando ogni riga.

ps [opzioni]

stato dei processi

DESCRIZIONE

ps visualizza diverse informazioni sullo stato dei processi attivi nel sistema. Il risultato della esecuzione del comando é una tabella che contiene sulle righe un nome di processo e sulle colonne informazioni relative al processo. Il contenuto delle colonne é precisato da una intestazione. La colonna marcata con UID contiene il numero identificativo dell'utente che ha lanciato il comando, il PID é il numero di processo, il PPID é il numero del processo padre, PRI é il livello di prioritá del processo, TIME é il tempo di esecuzione del processo, CMD é il nome del processo.

PRINCIPALI OPZIONI

-e visualizza informazioni per tutti i processi e non solamente per quelli dell'utente che ha mandato in esecuzione il comando.

-f vengono visualizzate tutte le informazioni che il comando puó fornire.

pwd

path-name directory corrente

DESCRIZIONE

pwd fornisce il path-name completo del directory corrente.

rm [-fri] file...

cancella file

DESCRIZIONE

rm cancella i file specificati negli argomenti, se esiste un solo link ai file. Qualora esistano molti riferimenti allo stesso file, creati con il comando ln, il file non è cancellato fisicamente, ma viene rimosso soltanto il riferimento nel directory corrente.

PRINCIPALI OPZIONI

-f se qualche file specificato tra gli argomenti è protetto in scrittura, rm chiede conferma prima di effettuare la cancellazione, a meno che non sia stata specificata l'opzione -f nel qual caso rm forza la cancellazione senza conferma.

-r rm -r directory cancella il directory indicato e ricorsivamente tutti i file contenuti in esso e nei suoi sottodirectory. L' opzione -r è l'unica che consenta di cancellare un directory con il comando rm.

-i viene richiesta una conferma prima della cancellazione dei file.

ESEMPI

rm -ri lettere

Esamina tutti i file a partire dal sottodirectory lettere del directory corrente. Dopo conferma li cancella.

rmdir directory ...

cancella directory

DESCRIZIONE

rmdir cancella i directory specificati negli argomenti. Un directory puó essere cancellato con questo comando solo se non contiene alcun file.

sort [-cmubdfinr] [-tcar] [+campo [-campo]] ... [-o file] file ...

ordina ed effettua merge di file

DESCRIZIONE

Se non é specificato alcun argomento sort ordina lo standard input in senso crescente, considerando come chiave di ordinamento l'intera riga. Specificando il nome di un file da ordinare, sort ordina quel file. Se sono indicati piú file, sort li ordina e ne effettua il merge. Specificando -o file_output, l'output di sort viene memorizzato sul file file_output invece di essere inviato sullo standart output. E' possibile effettuare il sort su una o piú chiavi definite a piacere. Ogni riga é pensata suddivisa in campi separati dal carattere specificato nella opzione -tcarattere del comando. Ogni campo é composto da un certo numero di caratteri. Il comando

```
sort -tcarattere +interol.intero2 -intero3.intero4
filel
```

ordina filel. Ogni riga del file é divisa in campi separati dal carattere scritto dopo il -t. Il primo carattere della chiave di ordinamento si ottiene saltando interol campi e poi intero2 caratteri dall'inizio della riga. L'ultimo carattere della chiave di ordinamento é immediatamente precedente a quello che si ottiene saltando intero3 campi e poi intero4 carat-

teri dall'inizio della riga. E' possibile ordinare secondo molteplici chiavi, specificandone il carattere di inizio e di termine. Se viene omessa l'indicazione sul carattere finale della chiave si assume che termini a fine linea. La specifica delle chiavi di ordinamento può essere seguita da uno o più flag scelti tra i caratteri bdfinr il cui significato è precisato nelle opzioni.

OPZIONI PRINCIPALI

-c esegue soltanto il controllo che il file sia ordinato.

-m supponendo che i file di input siano già ordinati, sort ne effettua il merge.

-u invia in output una sola copia delle righe duplicate.

-b non considera i caratteri bianchi iniziali delle chiavi di ordinamento.

-r ordinamento discendente invece che ascendente.

-tcarattere specifica il carattere separatore di campo. Per default i campi sono separati dal carattere spazio.

split [-n] [file1 [nome]]

divide un file in parti

DESCRIZIONE

split genera un insieme di file, ciascuno comprendente soltanto n righe del file specificato nell'argomento file1. n può essere omesso, in tal caso il file è spezzato in parti costituite da 1000 righe. I file generati hanno i nomi nomeaa, nomeab, nomeac, ...,nomezz. Qualora nome non venga specificato viene assunto come default x. I nomi dei file contenenti le parti del file da spezzare saranno cioè xaa, xab, xac, ..., xzz.

ESEMPI

split -100 lungofile

Spezza il file lungofile in pezzi creando i nuovi file xaa, xab, ...

split -100 < lungofile

L'effetto é il medesimo del comando precedente ma il nome del file da spezzare non viene fornito come argomento. Invece viene effettuata una redirezione dell'input.

su [-] [utente [argomenti...]]

diventa il super user o un altro utente

DESCRIZIONE

su consente di diventare il super user del sistema senza effettuare il logout del sistema e rientrare come root. Se viene specificato un nome di utente, su permette di diventare l'utente specificato. In entrambi i casi chi esegue il comando deve conoscere la password di login del super user o del secondo utente. Il comando manda in esecuzione un nuovo processo SHELL per interpretare i comandi, quindi per ritornare allo stato precedente occorre battere un EOF (control D), analogamente a quanto si fa per effettuare il logout dal sistema. L'EOF segnala al nuovo SHELL che non vi é alcun altro comando da leggere.

ESEMPI

su

Diventa super user.

sync

aggiorna il superblocco

DESCRIZIONE

Il comando manda in esecuzione la chiamata di sistema dello stesso nome, la quale aggiorna il superblocco. E' necessario dare tale comando quando si vuole avere la certezza che la situazione attuale sia stata salvata su memoria di massa. Il comando va quindi comunque eseguito, per salvaguardare l'integritá del file system, quando si decide di spegnere il sistema.

tail [+- [num][lbc[f]]] [file]

visualizza la "coda" di un file

DESCRIZIONE

tail visualizza la parte terminale di un file. Il comportamento di default é quello di visualizzare le ultime 10 righe del file. E' possibile vedere le ultime n righe, blocchi o caratteri del file specificando -numerol, -numerob, -numeror. In alternativa é possibile visualizzare il file a partire dell'n_esima linea, blocco o carattere specificando tail +numerol, +numerob, +numeror.

ESEMPI

```
tail -50c lettera
```

Visualizza gli ultimi 50 caratteri del file lettera.

tee [-i] [-a] [file]...

derivazione da un pipe

DESCRIZIONE

tee copia il suo input sull'output e sul file specificato. Inserito in una catena di comandi collegati da pipe, consente di memorizzare su un file i dati che fluiscono nel pipe.

OPZIONI PRINCIPALI

-a l'output di tee viene appeso al file specificato nell'argomento.

ESEMPI

```
ls -l | tee mieifile | wc -l
```

L'elenco dei file del directory corrente prodotto da ls -l costituisce l'input di tee che lo passa a wc. wc contando le righe, conta il numero di file presenti nel directory corrente. tee permette di generare una derivazione nel pipe, l'elenco dei file viene anche salvato sul file mieifile.

test espressione
[espressione]

valuta espressioni booleane

DESCRIZIONE

test valuta le espressioni costruite sulla base degli operatori seguenti e ritorna un exit value 0 (vero) o diverso da 0 (falso). Questo comando viene normalmente usato nelle procedure shell in connessione alle istruzioni if e while. Gli operatori di uso più comune sono:

-r f vero se il file f esiste ed é leggibile

-w f vero se il file f esiste ed é scrivibile

-x f vero se il file f esiste ed é eseguibile

-f f vero se il file f esiste ed é un file normale

-s f vero se il file f esiste ed é ha lunghezza maggiore di zero

-z s vero se la lunghezza della stringa s é zero

-n s vero se la lunghezza della stringa s é maggiore di zero

s1 = s2 vero se le stringhe s1 e s2 sono identiche

s1 != s2 vero se le stringhe s1 e s2 non sono identiche

s vero la stringa s non é la stringa nulla

n1 -eq n2 vero se gli interi n1 e n2 sono uguali. In modo simile operano i flags -ne, -gt, -ge, -lt, -le.

Possono essere utilizzati i seguenti operatori per combinare i termini primari precedenti:

! operatore di negazione unaria

-a operatore and binario

-o operatore or binario

( ) per raggruppare le espressioni e modificare le priorità di esecuzione

PRECAUZIONI

Tutti gli atomi delle espressioni debbono essere argomenti separati l' uno dall' altro. Inoltre occorre citare con \ o '' le parentesi in quanto altrimenti verrebbero interpretate dallo shell.

Quando si usa la seconda forma del comando, quella delimitata da parentesi quadre [ ], queste debbono essere delimitate da spazi.

umask [protezioni]

protezioni di default per file e directory

DESCRIZIONE

umask permette di specificare quali protezioni devono essere attivate all'atto della creazione di un file. I file vengono creati per default in modo ottale 666 ovvero con permesso di lettura, scrittura ed esecuzione da parte del proprietario, gruppo e resto degli utenti. I directory vengono creati in modo ottale 777 ovvero con permesso di lettura, scrittura e ricerca per il proprietario, gruppo e utenti rimanenti. Dopo aver eseguito il comando umask, il valore di default delle protezioni di file e directory all'atto della loro creazione viene modificato. Per ottenere il nuovo default, più restrittivo, occorre sottrarre le cifre della maschera alle cifre corrispondenti di 777 oppure 666. Qualora l'argomento non venga specificato umask visualizza il valore corrente della maschera.

ESEMPI

umask 066

I file e directory vengono creati senza permesso di scrittura e lettura per gruppo e resto degli utenti. I file vengono creati in modo 600 ed i directory in modo ottale 711.

uniq [-udc [+n] [-n]]] [input [output]]

determina le righe non ripetute

DESCRIZIONE

Se non viene specificato alcun argomento, uniq invia sullo standard output lo standard input, generando peró una sola copia delle righe eguali e consecutive presenti. Se vengono specificati gli argomenti input e output, uniq agisce su tali file.

OPZIONI PRINCIPALI

-u nessuna copia delle righe duplicate é visualizzata.

-d vengono visualizzate le sole righe duplicate.

-c vengono visualizzate le righe non duplicate ed una sola copia delle duplicate, come avviene se non é specificata alcuna opzione, ma ogni linea é preceduta dal numero di volte in cui essa appare ripetuta nel testo.

ESEMPI

sort miofile ¦ uniq -c determino le righe ripetute anche se non sono consecutive.

wc [-lwc] [file...]

conta le parole di un file

DESCRIZIONE

wc conta il numero di caratteri, linee e parole presenti nei file specificati dagli argomenti. Qualora non sia specificato alcun file, wc legge lo standard input.

OPZIONI PRINCIPALI

-l conta le sole linee.

-w conta le sole parole.

-c conta i soli caratteri.

ESEMPI

```
ls -l | wc -l
```

Determina il numero di file contenuti nel directory corrente.

who [-uTlpdbrtas] [file_who]
who am i

visualizza gli utenti attivi

DESCRIZIONE

who fornisce il nome, terminale ed orario di login di tutti gli utenti collegati al sistema. who am i fornisce informazioni sull'utente che esegue il comando. Se come argomento file_who é specificato il nome del file /etc/wtmp, viene riportata una traccia di tutti i login e logout dal sistema a partire dalla data di creazione di quel file.

write nome-utente [nome-terminale]

scrive a un altro utente

DESCRIZIONE

Questo comando consente di scrivere ad un altro utente attualmente collegato. Dopo avere mandato al ricevente un messaggio che lo avvisa della cosa, e gli indica da chi viene la comunicazione e da quale terminale, write preleva le righe dal terminale da cui é stato invocato, e le manda al terminale dell'utenrte ricevente. Quando chi deve ricevere il messaggio é alloggiato a piùdi un terminale, cosa consentita in ambiente UNIX, il comando write viene inibito, a meno che non si specifichi l'altrimenti opzionale nome-terminale.

PRINCIPALI OPZIONI

-l viene riportato l'elenco dei terminali liberi.

-b viene indicata la data e l'ora dell'ultimo boot del sistema operativo.

-t viene indicata la data e l'ora dell'ultimo cambiamento dell'ora del sistema.

ESEMPI

```
who ¦ grep stefano
```

Determina se stefano sta usando il sistema.

Il sistema operativo UNIX si avvia a diventare uno standard per calcolatori a 16 bit anche nel mondo industriale, così come è già avvenuto nel mondo universitario. Questo libro, che nasce dall'esperienza diretta in quell'ambito, si presenta come una introduzione didattica al mondo di UNIX, e conduce per mano il lettore a toccarne le più fondamentali funzionalità. Per la sua completezza, esso si porge come un vero e proprio manuale per chi voglia fattivamente operare in questo ambiente di punta dell'informatica contemporanea.

Maurizio Matteuzzi è titolare di "Filosofia del Linguaggio" presso il Dipartimento di Filosofia dell'Università di Bologna. Si occupa di linguaggi formali e di logica simbolica.

Paolo Pellizzardi è tra i soci fondatori della West 80 di Bologna. Si è occupato attivamente di sviluppo del software, sia di base che applicativo, e delle problematiche ad esso relative.

L. 19.000 Cod. 543P ISBN 88-7056-221-2

186
Ambiente Unix
Maurizio Matteuzzi
Paolo Pellizzardi
GRUPPO
EDITORIALE
JACKSON